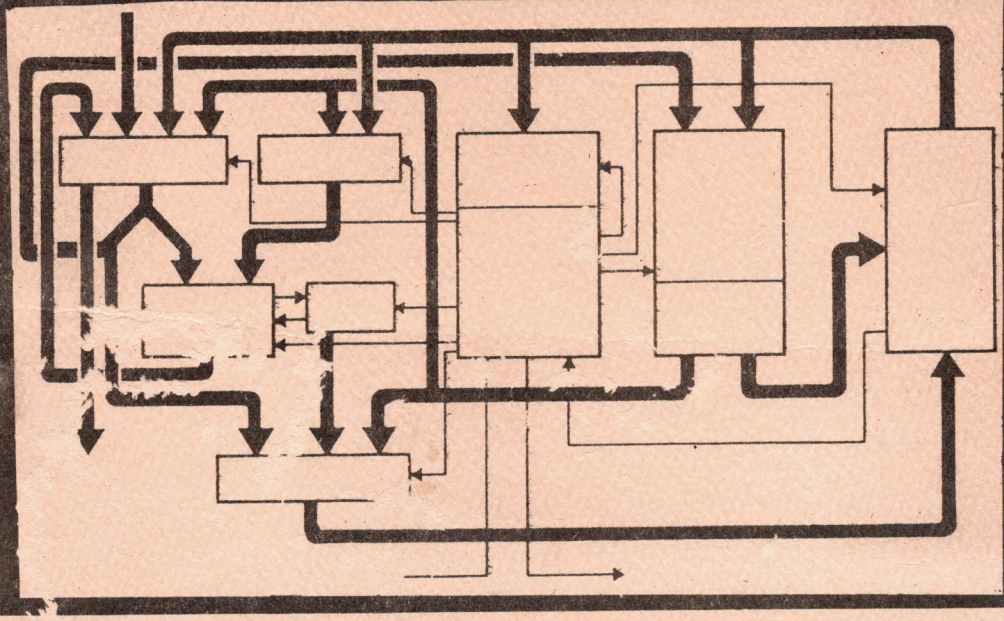
Val Hobbs

MURDOCH UNIVERSITY

M216

COMPUTER ARCHITECTURE



EXTERNAL STUDIES UNIT

MURDOCH UNIVERSITY

EXTERNAL STUDIES UNIT

– M216 –

COMPUTER ARCHITECTURE

First Semester 1989

STUDY GUIDE and COURSE MATERIALS

Course Coordinators

1988 Dr D.P. Hodgson

1989 Dr N.K. Sharda

JOB 702/89

M216 : COMPUTER ARCHITECTURE

Introduction

Objectives

The objectives of the course are for the student to demonstrate:

(a) an understanding of logic design and the organisation of hardware components of computers;

(b) an introductory knowledge of computer architecture;

(c) knowledge of the representation and control of information within a computer;

(d) knowledge of and skills in the use of machine language.

The prerequisite for M216 *Computer Architecture* is M104 *Principles of Computer Science.*

The Coordinator for *Computer Architecture* is Dr Nalin Sharda, located in the Science and Computing Building (EAIII), Room 2-22, telephone (09) 332 2798, or messages to the Computer Science secretary 332 2521.

Textbook

Mano, M. Morris *Computer Engineering: Hardware Design* Prentice-Hall 1988, ISBN 0-13-162710-4

Documentation for the computer system that you will use for your practical work, Disk Operating System manual etc.

e.g. *Microsoft MS-DOS User's Guide and User's Reference* Microsoft (version 2.0 or later.)

Wolverton, Van *Running MSDOS* 2nd ed. Microsoft Press (Penguin Books) X 005.446 W869 158842E

Norton, P. *Programmers Guide to the IBM PC* Microsoft Press (Penguin Books) 1985

Study Materials and Equipment.

The following items are required for this course:

(a) This Study Guide

(b) The required text book, and manuals for the computer system you will be using.

(c) *Computer System* **The student is responsible for arranging access to a suitable computer system** either by attendance at the University, providing your own, or arranging with your employer or a regional college to have good access.

The standard system assumed will be

Hardware	An IBM PC or compatible
Software	PC/MS-DOS 2.0 or higher Debug Utility
Documentation	Manuals for the Operating System, Debug Utilities

(d) **Materials to be loaned by the University**

(i) A digital logic kit to be used for experiments.

(ii) Debug Utility (if you require).

Please complete and return to External Studies, as soon as possible, the tear-out loan form at the end of the buff pages. Where possible, nominate personal collection, bringing the completed form with you.

Study Arrangement

During the first week of semester you will be allocated a Tutor who will guide your study, provide advice, and mark your assignments.

The course will use IBM PC or compatible microcomputers. Access to a personal computer is either by attending on campus or arranging access to a machine by other means.

You must supply details of the system you intend to use to your tutor. Do this by attaching a letter to your first assignment submission. Details on your proposed access to computing facilities were collected at enrolment time, but the letter to your tutor is required for confirmation and update.

If you do not have access to a system of the specified type, either independently or by campus attendance, you should lodge a withdrawal from the course without delay. If you have limited access you should make sure that you will have sufficient time to complete all the required practical and assignment work.

Studying the Course: Study Pace and Methods

In 1989 the offering of M216 *Computer Architecture* is option A and E only (one semester pace). Try to keep up with the weekly practice sessions.

Submission of Coursework

Each assignment should be sent, together wih an Assignment Attachment, to the External Studies Unit. The recommended timetable for submissions is given in the section on practice sessions (section C).

An Assignment will not normally be accepted if submitted after the due date for the following assignment. For example Assignment 1 will not be accepted after the due date for Assignment 2.

Requests for later submission must be made to your tutor in writing with reasons for the extension.

The final deadline date for submission of Assignments is firmly fixed:

Option A,E : Friday 9 June 1989

You should try and follow this timetable as closely as possible to even out the workload for you and your Tutor. **Batches containing more than two assignments in one submission are not acceptable and the excess will be returned to you unmarked for later re-submission.** It is essential that you get feedback from your Tutor at regular intervals.

To submit a program for assessment you should supply a listing and a 5.25 inch floppy disk (DOS format) containing the program in source and compiled form. The first page of your Assignment must contain a printed or neatly written list of the files on your disk, with an identifying description for each file, including reference to the question number, e.g.

Name of Program File	e.g. Contfile.asm
Name of Sample input data file	e.g. Contfile.i
Name of Sample output data file	e.g. Contfile.o
Name of Documentation File	e.g. Contfile.doc
Name of Batch file to run program	e.g. Contfile.bat

Your assignment and disk submissions are made through the External Studies Unit, and each submission must have an Assignment Attachment. Ensure that you include a listing of the filenames which relate to the current assignment. The 5.25 inch floppy disk may be protected for postal despatch by enclosing within 3 or 4 pieces of stiff cardboard, cut to fit the usual despatch envelopes. Never post an unprotected disk because the severe bending which may occur in transit may result in an unreadable disk. Advice on the recovery of damaged disks may be obtained from External Studies.

Ensure that you keep a backup copy of each disk which you have submitted, and that **each disk is labelled with your name, student number, and course code M216.**

Although your Tutor will be marking your Assignments as quickly as possible, you should not wait for the return of a submission before commencing study for the next submission

Assessment

In order to pass you must achieve a satisfactory performance in each of the Assignment and Examination components of the course.

(a) *Assignment assessment*

Each assignment will be classified as Satisfactory, Borderline, or Fail and will contribute to your overall performance in the Assignment component of assessment.

You may be required to undertake further work if you have not performed satisfactorily in an assignment considered as critical to the course.

The final date for submission of assignments is given in the table of assignment submissions given above. Extensions beyond this date are not available except through application to the University Administration in accordance with the deferred assessment procedures described in Degree Regulation 19.

(b) *Examination assessment*

The final examination will be held during the Assessment Period 19-30 June. The University's Examination Officer will contact students in advance in order to arrange supervised off-campus examinations for students unable to attend the on-campus venue.

Course grades will normally be awarded on the basis of your overall marks using the approximate scale:

A >= 80 marks

80 > B >= 65 marks

65 > C >= 50 marks

Fail < 50 marks

Overall marks will be made up of the following components:

Submissions	30%
Written Examination	70%

Previous examination papers are given at the end of the study guide. Questions in the examination will be generally similar to the questions encountered in the assignments. Since the text book has changed from the previous years, this year's questions could be quite different. The examination will be "closed book", and scheduled for three hours preceded by ten minutes reading time.

Reference Books

Library holdings in computer architecture are under development. However, some of the following may provide useful optional reading.

Foster, Caxton C. *Computer Architecture* 2nd ed. Van Nostrand 1976	X 621.39 F754 070718B
Kuck, David J. *The Structure of Computers and Computations* Wiley 1978	X 004.2 K95 087210C
Atkin, J.K. *Computer Science* 2nd ed. Macdonald and Evans 1980	X 004 A873 133528B
Dowsing, Roy and Woodhams, Frank *Computer Architecture : a first course* Van Nostrand Reinhold 1985	X 004.22 D752 654359K
Lippiatt, Arthur G. and Wright, Graham G.L. *The Architecture of Small Computer Systems* 2nd ed. Prentice-Hall 1986	X 621.392 L765 158278B
Bartee, Thomas C. *Digital Computer Fundamentals* 4th ed. McGraw-Hill 1977	X 004 B283 069717A
Mano, M. Morris *Digital Design* Prentice-Hall 1984	X 621.395 M285 142481J
Tanenbaum, Andrew S. *Structured Computer Organization* 2nd ed. Prentice-Hall 1984	X 005.1 T164 150057B
Abel, Peter *Assembler for the IBM PC and PC-XT* Reston Publishing 1984. ISBN 08359 0110 6	[On order]
Aspinall, David (ed.) *The Microprocessor and its Application: An Advanced Course* Cambridge University Press 1978	X 004.256 M626 109458J

Colin, Andrew J.T. *Fundamentals of Computer Science* The Macmillan Press 1980 — X 004 C696 128996J

Horowitz, Paul and Hill, Winifield *The Art of Electronics* Cambridge University Press 1980 [approx $45] good 1st year book on analog and digital useful examples. — X 621.381 H816 651772F

Leventhal, Lance A. *Microcomputer experimentation with the IBM PC* Holt Rinehart Winston 1988 ISBN 0 03 009542 5 — [on order]

Liu, Yu-Cheng and Gibson, Glenn A. *Microcomputer systems: The 8086/8088 family: architecture, programming and design* Prentice-Hall 1986. ISBN 0 13 581851 6 025 — X 004.256 L783 149001G

Osborne, Adam *An Introduction to Microcomputers: Basic concepts* 1980 — X 004.16 O81

Vol.0, *The Beginners Book* — 128178J [Vol.0]

Vol.1, 2nd ed. Osborne McGraw-Hill — 128177D [Vol.1]

Vol.2, *Some Real Microcomputers*

Vol.3, *Some Real Support Devices*

Uffenbeck, John *The 8086/8088 family: design, programming and interfacing* Prentice-Hall 1987 ISBN 0 13 246638 4 — [on order]

Access to On Campus PC XT's for Computer Architecture

Contact External Studies to obtain a Security Clearance, and details on access hours, etc.

Loans of Logic Kits and Software Disks

Please complete and return to External Studies, as soon as possible, the tear-out loan form at the end of the buff pages. Where possible, nominate personal collection, bringing the completed form with you.

Instructions on the use of this equipment are provided in the notes relating to practical work in this booklet, and with the kit itself. Note that you should not remove any chips from the board. If battery power is exhausted, you purchase and fit a new battery. The Computer Science technical staff (09 332 2802) will provide advice if you suspect that a component failure has occurred.

Please return the kit promptly, through External Studies, at the end of your enrolment period.

COMPUTER SCIENCE PROGRAMME

Conditions for Use of CSP Computer Systems

1. Any account information provided is for your exclusive use and may not be given to another person.

2. The access to CSP facilities is to be used only for your own work on the projects assigned in the courses or as agreed with the Computer Science Programme and may not be spent on hobby computing or private consulting work.

3. You are responsible for the management of your own account.

   (a) The login-name and password (if applicable) should NEVER be given to anyone else.

   (b) The password should be changed frequently (if applicable).

   (c) You are responsible for the security of your files. Use the recommended backup procedures regularly.

4. Software products available on the computers are proprietary (unless stated otherwise). If you use any software other than your own, it is done in confidence. Such software may not be copied, given, or sold to any other person.

5. Any misuse of computer software or hardware will result in termination of your access to the computer systems.

Dr Roger Atkinson
External Studies Unit
Murdoch University
MURDOCH 6150

Dear

Request for equipment loan

For the purposes of my study in the course

M216 Computer Architecture Option E

I request loan of the following items: ESU staff - record no

Digital Logic kit and components _____

_____ _____

_____ _____

Nominate either:

Personal collection by myself or agent _____

or

despatch by Australia Post

loan period *20 Feb 1989* to *9 June 1989*

I undertake responsibility to use the loaned equipment for study purposes in accordance with the instructions provided; to return the equipment in good order on or before the end of the specified loan period; and I further undertake responsibility for any loss or damage incurred through negligence on my part.

Signature _____ Student No _____ Date _____

Name _____ Your address should be

Address _____ the same as in University

(print) _____ Records

Authorised _____

Deposit to be enclosed by student: $Not Required (cheque to Murdoch Uni)

Issue/despatched _____ Date _____

Return recorded by _____ Date _____

Deposit return requisition by _____ Date _____

Account Coding _____

COMPUTER SCIENCE PROGRAMME

Application for access to Computer Science Computers

Full Name .....

Student Number .....

Home Address .....

.....

..... Area Code .....

Telephone Number: Home ..... Work .....

1st Programme ..... 2nd Programme .....

Course Name and Number .....

Course Enrollment option (A,B,E,G,F,H) .....

Please indicate which computer system(s) you want to use

LSI 11/73 ☐ MacLab ☐ PCLab ☐

Please list the times at which you want to access the system(s)

Declaration: I have read the conditions of Use and agree to abide by them

Signed ..... Date .....

You have been granted access to:

User-ID ..... Password .....

Authorised by ..... Date .....

COURSE ORGANIZATION

The course is organized in two parts. The first part covers the topics which are applicable to computers in general. The second part is based on a specific processor, i.e. the Intel 8086/8088 CPU.

PART I

Topics

1. Binary Numbers and Codes
2. Digital Circuits
3. Combinational Systems
4. Sequential Logic
5. Registers and Counters
6. Memory and Programmable Logic
7. Register Transfer and Computer Organization
8. Control Logic Design
9. Computer Instructions and Addressing Modes
10. Design of a Central Processing Unit
11. Input-output and Communication
12. Memory Management

Text: M. Morris Mano *Computer Engineering, Hardware Design* Prentice-Hall 1988

Ref: Section D and E of the notes.

PART II

1. 8086/8088 Architecture
2. Addressing Modes
3. Instruction Set
4. Debug Commands
5. 8086 Programming

Text None

Ref: Sections **F, G, H, I, J** and **K** of the notes

Weekly Sessions:

The following practice sessions specify the topics for reading on weekly basis and include problem to be solved as well as practical work. There are 12 sessions in all. Practice session number x is to be done in week number x. This leaves week 13 free for catching up with any pending work. Therefore the final deadline for submission is fixed at 9-6-89.

Problem Solving:

A list of problems is given for the student to solve. By solving these problems the student will get further insight into the subject matter. These will also prepare the student for the written examination. Problems marked with an asterisk(*) will be checked more thoroughly by the tutor.

Internal mode: Problems listed for each session should be attempted by the internal student prior to coming for the tutorial/laboratory class and the attempted solutions shown to the tutor. An **A4 size notebook or file** must be maintained in which all these solutions are recorded.

External mode:The external students should submit the solutions by the specified date. Use **A4 size paper** and keep these in a file. This file will come handy for revision at examination time.

Practical Work:

Practical work is designed to give the student hands-on experience with digital electronic components and machine language programming.

An **A4 size 'report file'** should be maintained for recording the results of the practical work under the following headings:

Practical Session No:

Topic:

Procedure: This should include the main steps involved, truth tables, diagrams, printouts etc.

Discussion and Conclusions:

The report file can be an A4 size notebook or a folder containing A4 size paper. The reports must be brief and **neat**. Each report must start on a new page.

This file needs to be submitted twice during the semester. First submission will be after completing the set of digital design exercises and the second submission will be after completing the set of programming exercises; for the internal as well as the external students.

Important:

A copy of the following page **must** be attached to the cover page of each internal submission. All details must be entered. Assignment title refers to the title of the submission, e.g. Digital Logic Exercises, 8086/8088 Programming etc. Make sure you enter your name as specified in the enrollment documents. The external students should observe the above mentioned points in filling the assignment attachments.

****

SUBJECT **COMPUTER ARCHITECTURE M216**

TUTOR _____

ASSIGNMENT TITLE _____

DATE - DUE ON _____

DATE - SUBMITTED ON _____

STUDENT NAME

_____

(FIRST) (SECOND INITIAL) (LAST)

STUDENT NO. _____

GRADE _____

****

٢

PRACTICE SESSION NUMBER 1

Topics for reading:

Binary Numbers and Codes; from chapter 1 of the text book by Mano.

Problem Solving:

Submit solutions to the following problems from the text book by Mano, chapter 1.

Questions: 1.1, 1.2, 1.3, 1.8*, 1.9, 1.12, 1.14, 1.16, 1.19, 1.23*, 1.27*, 1.30, 1.31.

Note : Pay special attention to the questions marked with an asterisk(*).

Submission:

External : By 24-2-89 .
Internal : Show to your tutor in week 2 .

Practical Work:

Digital Logic Exercise No.1, from section D of the study guide.

The aim of this exercise is to gain familiarity with the Digital Trainer Kit. Make sure that you understand the function and operation of each part of the kit. If there are any doubts clear these with your tutor.

Submission: By 21-4-89, both internal and external modes.

PRACTICE SESSION NUMBER 2

Topics for reading:

Digital Circuits; from chapter 2 of the text book by Mano.

Problem Solving:

Submit solutions to the following problems from the text book by Mano, chapter 2.

Questions: 2.1 a&b*, 2.2 a&b, 2.3 a&b, 2.5*, 2.10*, 2.11a&b, 2.13 b, 2.15, 2.19 a, 2.24*.

Note : Pay special attention to the questions marked with an asterisk(*).

Submission:

External : By 3-3-89 .
Internal : Show to your tutor in week 3 .

Practical Work:

Digital Logic Exercise No.2, from section D of the study guide.

The aim of this exercise is to understand the working of TTL (transistor-transistor-logic) integrated circuits.
Learn the use of an LED as a logic probe.

Submission: By 21-4-89, both internal and external modes.

PRACTICE SESSION NUMBER 3

Topics for reading:

Combinational Systems: from chapter 3 of the text book by Mano.

Problem Solving:

Submit solutions to the following problems from the text book by Mano, chapter 3.

Questions: 3.1, 3.4*, 3.7, 3.10, 3.14*, 3.15, 3.21*, 3.23, 3.26, 3.38.

Note : Pay special attention to the questions marked with an asterisk(*).

Submission:

External : By 10-3-89 .
Internal : Show to your tutor in week 4 .

Practical Work:

Digital Logic Exercise No.3, from section D of the study guide. .

The aim of this exercise is to construct and test some combinational logic circuits.

Submission: By 21-4-89, both internal and external modes.

PRACTICE SESSION NUMBER 4

Topics for reading:

Sequential Logic; from chapter 4 of the text book by Mano.

Problem Solving:

Submit solutions to the following problems from the text book by Mano, chapter 4.

Questions: 4.1, 4.2*, 4.5, 4.11, 4.14*, 4.16, 4.27*.

Note : Pay special attention to the questions marked with an asterisk(*).

Submission:

External : By 17-3-89 .
Internal : Show to your tutor in week 5.

Practical Work:

Digital Logic Exercise No.4, from section D of the study guide.

The aim of this exercise is to construct and test the operation of flipflops.

Submission: By 21-4-89, both internal and external modes.

PRACTICE SESSION NUMBER 5

Topics for reading:

Registers and Counters; from chapter 5 of the text book by Mano.

Problem Solving:

Submit solutions to the following problems from the text book by Mano, chapter 5.

Questions: 5.1, 5.6, 5.7*, 5.10, 5.18*, 5.22, 5.30*.

Note : Pay special attention to the questions marked with an asterisk(*).

Submission:

External : By 24-3-89 .
Internal : Show to your tutor in week 6 .

Practical Work:

Digital Logic Exercise No.5, from section D of the study guide.

In this exercise you will work with a 4-bit binary counter.

Submission: By 21-4-89, both internal and external modes.

PRACTICE SESSION NUMBER 6

Topics for reading:

Memory and Programmable Logic; from chapter 6 of the text book by Mano.

8086/8088 Architecture; from sections 2.1 and 2.2 of the book by Liu and Gibson, given in section F of the study guide.

Problem Solving:

Submit solutions to the following problems from the text book by Mano, chapter 6.

Questions: 6.1*, 6.2, 6.5*, 6.10, 6.17*, 6.21, 6.25*.

Submit solution to following problems from the book by Liu and Gibson, chapter 2, page 51(of the book), given in section F of the study guide.

Questions: 1*, 2*.

Note : Pay special attention to the questions marked with an asterisk(*).

Submission:

External : By 7-4-89 .
Internal : Show to your tutor in week 7 .

Practical Work:

Digital Logic Design Exercise, from section D of the study guide. This exercise can be done over two weeks.

In this exercise you will put to use the skills acquired in the previous exercises. Read the section on fault finding and follow the hints given for neat wiring. Internal students will show the operation of the circuit to the tutor in the laboratory while the external students will send back the circuit wired on the digital trainer kit.

For the external students it is important that your wiring is good enough to survive the postal system's shocks. Your tutor will only check if the circuit works as intended, and will not try to do any fault finding for you.

Submission: By 21-4-89, both internal and external modes.

PRACTICE SESSION NUMBER 7

Topics for reading:

Register Transfer and Computer Operations; chapter 7 of the text book.

8086/8088 Addressing Modes, Instruction Formats, and Execution Timing; sections 2.3,2.4 & 2.5 of the book by Liu and Gibson, given in section F of the study guide. Also, section 2.3 of the book by Uffenbeck, given in section G of the study guide.

Problem Solving:

Submit solutions to the following problems from the text book by Mano, chapter 7.

Questions: 7.1, 7.2*, 7.11*

Submit solutions to the following problems from the book by Liu and Gibson, chapter 2, page 51(of the book), given in section F of the study guide.

Questions: 3, 7, 8a*, 8d.

Note : Pay special attention to the questions marked with an asterisk(*).

Submission:

External : By 14-4-89 .
Internal : Show to your tutor in week 8.

Practical Work:

Digital Logic Design Exercise, from section D of the study guide. Continued from week 6.

Submission: By 21-4-89, both internal and external modes.

PRACTICE SESSION NUMBER 8

Topics for reading:

Control Logic Design; chapter 8 of the text book by Mano.

8086/8088 Data Transfer Instructions; section 2.4 of the book by Uffenbeck, given in section G of the study guide.

Problem Solving:

Submit solutions to the following problems from the text book by Mano, chapter 8.

Questions:8.1*, 8.2, 8.4, 8.6*.

Submit solutions to the following problems from the book by Uffenbeck, section 2.4.

Problems: 2.4.(1-6)* , page 60(of the book), given in section G of the study guide.

Note : Pay special attention to the questions marked with an asterisk(*).

Submission:

External : By 21-4-89 .
Internal : Show to your tutor in week 9.

Practical Work:

Debug Utility, given in section H of the study guide.

The debug utility comes with the disk Operating System (DOS). Its function is to let you work closely with the system hardware. Since it gives you access to the systems low level facilities **it becomes your responsibility to ensure that you do not crash the system**. Debug commands are given in section H of the study guide. Learn these commands and try out their operation.

Record the work done in your report file. Include relevant printouts. Note that you will get cleaner printouts by using the Ctrl-P command rather than by using Shift-PrintScreen. Highlight the important parts of the printout. You may also write comments on the printout. This applies to all sessions that follow.

Submission: By 9-6-89, both internal and external modes.

PRACTICE SESSION NUMBER 9

Topics for reading:

Computer Instructions and Addressing Modes; from chapter 9 of the text book by Mano.

8086/8088 String Instructions; from section 2.5 of the book by Uffenbeck, given in section G of the study guide.

Problem Solving:

Submit solutions to the following problems from the text book by Mano, chapter 9.

Questions:9.1, 9.2*, 9.3, 9.7*, 9.8*, 9.11, 9.17*, 9.28, 9.29.

Submit solutions to the following problems from the book by Uffenbeck, section 2.5.

Problems: 2.5.(1-4)* , page 65(of the book), given in section G of the study guide.

Note : Pay special attention to the questions marked with an asterisk(*).

Submission:

External : By 28-4-89 .
Internal : Show to your tutor in week 10.

Practical Work:

Machine Language Programming.

In this exercise you will learn how to enter and execute machine language programs. Read chapter 2 of the book by Peter Able, given in section I of the study guide. Do the working examples given in this section. Record the work done in your report file. Also attempt the questions given at the end of this chapter. Answer to selected questions are provided on the last page of section I .

Internal students should read through this section before coming to the laboratory, so that the experimental part can be finished in two hours.

Submission: By 9-6-89, both internal and external modes.

PRACTICE SESSION NUMBER 10

Topics for reading:

Design Of A Central Processing Unit; from chapter 10 of the text book by Mano.

8086/8088 Logical Instructions, from section 2.6 of the book by Uffenbeck, given in section G of the study guide.

Problem Solving:

Submit solutions to the following problems from the text book by Mano, chapter 10.

Questions:10.3, 10.4*, 10.8.

Submit solutions to the following problems from the book by Uffenbeck, section 2.6.

Problems: 2.6.(1-5)* , page 72(of the book), in section G of the study guide.

Note : Pay special attention to the questions marked with an asterisk(*).

Submission:

External : By 19-5-89.
Internal : Show to your tutor in week 11.

Practical Work:

Processing Data Arrays.

This session is based on laboratory 6 from the book by Lance A Leventhal and is given in section J of the study guide.
Do the problems and run the example programs given in this section.

Internal students should read through this section before coming to the laboratory, so that the experimental part can be finished in two hours.

Submission: By 9-6-89, both internal and external modes.

PRACTICE SESSION NUMBER 11

Topics for reading:

Input - Output and Communication, from chapter 11 of the text book by Mano.

8086/8088 Arithmetic Instructions, from section 2.7 of the book by Uffenbeck, given in section G of the study guide.

Problem Solving:

Submit solutions to the following problems from the text book by Mano, chapter 11.

Questions:11.2, 11.3*, 11.5*, 11.7, 11.13

Submit solutions to the following problems from the book by Uffenbeck, section 2.7.

Problems: 2.7.(1-8)* , page 85(of the book), given in section G of the study guide.

Note : Pay special attention to the questions marked with an asterisk(*).

Submission:

External : By 26-5-89.
Internal : Show to your tutor in week 12.

Practical Work:

8-Bit Arithmetic on the 8086/8088

This session is based on laboratory 9 from the book by Lance A Leventhal (pp 172 to 180) and is given in section J of the study guide. Do the problems and run the example programs given in this section.

Internal students should read through this section before coming to the laboratory, so that the experimental part can be finished in two hours.

Submission: By 9-6-89, both internal and external modes.

PRACTICE SESSION NUMBER 12.

Topics for reading:

Memory Management, from chapter 12 of the text book by Mano.

8086/8088 Transfer of Control and Process Control Instructions, from section 2.8 and 2.9 of the book by Uffenbeck, given in section G of the study guide.

Problem Solving:

Submit solutions to the following problems from the text book by Mano, chapter 12.

Questions:12.1, 12.3, 12.11*

Submit solutions to the following problems from the book by Uffenbeck, sections 2.8 and 2.9.

Problems: Odd number questions in the range 2.8.(1-11)* and 2.9.(1-3), given in section G of the study guide.

Note : Pay special attention to the questions marked with an asterisk(*).

Submission:

External : By 2-6-89.
Internal : Show to your tutor in week 13.

Practical Work:

16-Bit Arithmetic on the 8086/8088

This session is based on laboratory 9 from the book by Lance A Leventhal (pp 181 to 194) and is given in section J of the study guide. Do the problems and run the example programs given in this section.

Internal students should read through this section before coming to the laboratory, so that the experimental part can be finished in two hours.

Submission: By 9-6-89, both internal and external modes.

TEXT BOOKS

The following books will be the text for the course and must be acquired by every student.

1. M216 COMPUTER ARCHITECTURE STUDY GUIDE, N.K. Sharda and D.P. Hodgson, External Studies Unit, Murdoch University, 1989.

2. COMPUTER ENGINEERING: HARDWARE DESIGN, Morris M Mano, Prentice-Hall International Editions, 1988, ISBN 0-13-162710-4.

REFERENCE BOOKS

The following books are recommended for further reading. The relevant section from each one of these is included in the study guide.

1. THE 8086/8088 FAMILY: DESIGN, PROGRAMMING, AND INTERFACE, John Uffenbeck, Prentice-Hall International Editions, 1987, ISBN 0-13-246638-4.

2. MICROCOMPUTER SYSTEMS: THE 8086/8088 FAMILY ARCHITECTURE, PROGRAMMING, AND DESIGN, Yu-Cheng Liu and Glenn A. Gibson, Prentice-Hall International Editions, 1986, ISBN 0-13-581851-6 025.

3. MICROSOFT MS-DOS USER'S GUIDE AND USER'S REFERENCE, Microsoft Corporation, 1987.

4. ASSEMBLER FOR THE IBM-PC AND PC-XT, Peter Abel, Reston Publishing Company, 1984, ISBN 0-8359-0110-6.

5. MICROCOMPUTER EXPERIMENTATION WITH THE IBM-PC, Lance A. Leventhal, Holt Rinehart and Winston Inc., 1988, ISBN 0-03-009542-5.

INTRODUCTION :

This section contains a set of digital logic exercises designed to strengthen the understanding of digital systems. These will also provide hands-on experience with digital integrated circuits (ICs). A digital trainer kit will be used for the experiments. The kit comprises of solderless breadboards, toggle switches, push button switches, seven segment displays, clock signals, and power source. The kit to be used on campus runs off the mains power while the external mode kit runs off a 9 volt battery.

The required ICs are already in place on the kit. It is recommended that you **do not remove the ICs**. It is very easy to bend the IC pins while removing or inserting it into the breadboard.

IN CASE OF PROBLEMS :

If you suspect any IC or are having any other problem with the experiments, contact Dr. Nalin Sharda on (06)332-2798. In case you do not get him on the phone leave a message with the programme secretary, Ms Pauline Day on (06)332-2521.

TOOLS REQUIRED :

The experiments have been designed such that you can do these without needing any tools. Nevertheless the availability of a pair of nose pliers and a pair of cutter pliers would be of help. You can of course use your imagination to find alternatives. For example, to insert the wire ends into the breadboard a pair of eyebrow tweezers can comes handy.

Neat and systematic wiring is the sure way to a working circuit. Read the next section for hints on neat wiring.

TECHNIQUES FOR FAULT FINDING :

It is a rare thing for a circuit to work the first time you wire it. Given a faulty circuit it has to be checked step-by-step, the fault located and corrected.

If your circuit wiring is neat and systematic you have a better chance that the circuit may work the first time. Even if the circuit does not work the first time, neat and systematic wiring will make it much easier to find the fault.

To get a neatly wired circuit observe the following:

1. Route the wires on the breadboard close to the board's surface and use neat 90 degree bends.

2. Cut the wires to the required length, i.e. not too long.

3. Strip the insulation from the two ends no less than 4mm and no more than 6mm.

4. Insert resistors so as to lie flat on the board by cutting the two end leads to the required length and bending about 5mm of the ends at 90 degrees.

Being systematic involves observing the following points. (Or you can make your own set of rules and then stick to those rules).

1. Use as many different coloured wires as possible. This helps in tracing the connections for fault finding.

2. Designate wire colours, e.g. red for +5v (Vcc) and black for ground (GND) etc..

3. Make one of the top horizontal lines your Vcc bus and one of the bottom horizontal lines as the GND bus. Always make Vcc and GND connections to the respective bus. Never tap these voltages from one IC to another.

Even after observing all of the points mentioned above you may end up having a faulty circuit. Before explaining the procedure for fault finding I will list the most frequent faults.

Incorrect connection:
The IC pins are only 0.1 inch apart. In your first attempt at wiring a circuit, you can easily make a mistake. Always recheck your circuit connections before trying its operation.

Broken connection:
A connection from point A to point B may be broken due to:

a) the conductor being broken inside the insulation and therefore the break is not visible.

b) the wire end or an IC pin is not making good contact with the connection strip inside the breadboard.

Shorted connection:
If two conductors touch each other, when these are not meant to, a short circuit is formed. This can, in the worst case, burn out a component and will, at the least, cause a malfunction in the circuit. To avoid this problem make sure that you do not strip too much insulation at the wire ends.

FAULT FINDING:

Generally an instrument called a **logic probe** is used for fault finding in digital circuits . It is approximately the size of a marker pen and has a conducting tip. It can run off the 5v supply of the circuit under test. There are two or three LEDs on a logic probe. These LEDs indicate the presence of a HI, LO or a pulsing signal at the tip.

In the absence of a proper logic probe you can use one of the LEDs in your system as a logic probe. Assign one LED for this purpose and have a reasonably long wire connected to this LED (i.e. to the anode of the LED, the cathode is already connected to the ground bus). The free end of this wire can now be used for probing. If the LED turns ON, the probed point is a logic HI and if it remains OFF the probed point is either a logic LO or it is not connected. A pulsing signal can be observed only if the frequency of the signal is less than 0.5 Hz (approx.).

When you start checking a circuit always test the Vcc and the GND lines first. To locate a fault you can either start testing the circuit at the output and trace the circuit back towards the inputs or you can start at the inputs and trace the operation towards the outputs. You will have to make this choice depending upon the nature of the fault and the circuit under test.

One way to make sure that your circuit works the first time is to check its operation as you wire parts of the circuit. To do this in the following example first wire the NAND gate A [pins 1,2,3] and check that it operates as an inverter. Next wire the gates B [pins 4,5,6] and C [pins 12,13,11]. Check their operation for the different values of the inputs **a,b** and **x**. When you get correct operation from these gates then wire the gate D. You can now check operation of the complete circuit.

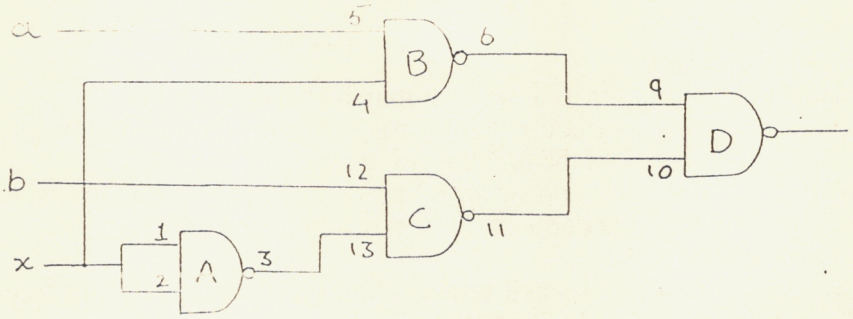


Figure D-0.0 WIRING DIAGRAM FOR THE 2 TO 1 MULTIPLEXER

If a circuit refuses to work and its operation does not seem to follow the logic it is implementing, even after a long session of fault finding, it is best to remove all connections and wire the circuit again. If you suspect the connection between an IC and the breadboard just press on the IC with your thumb and give it a bit of lateral pressure. **Always turn the power off for making any connections.**

Your kit includes either 1K ohm resistors or 1.2K resistors. In fact, for this kind of work any resistor between 1k and 10k will usually work. Some information on resistors and the colour code for these is given in figure D-0.1.

RESISTORS

They are generally small cylindrical devices between 5 and 40mm long. They are recognized by coloured rings around them.

Large wattage types are generally rectangular and have their values written on them.

Resistance is measured in ohms, which is abbreviated to Ohm or R. Thousands of ohms is abbreviated to K and millions of ohms to M.
Eg. 10,000 ohm is 10K
15,000 ohm is 15M

The coloured rings around a resistor determine its value. The first three indicate its resistance, and the fourth its tolerance.

Tolerance is how much it varies from its specified value. This is given as a percentage.

Eg. 5% tolerance. The fourth band is Gold. A 100 Ohm 5% tolerance resistor could be anywhere between 95-105 ohm.

Circuits generally take this into account.

To determine resistor value, the first two rings are the significant value, the third ring is the number of zeros following:-

Examples:-

68 ohms, 5% .... Blue, Grey, Black, Gold
1K ohm, 5%.... Brown, Black, Red, Gold
560K ohm, 5% Green, Blue, Yellow, Gold
47K ohm, 2% Yellow, Violet, Orange, Red

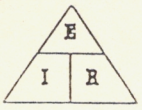
Ohm's Law

The relationship between resistance, voltage and current is defined by Ohm's Law. The formulas are -

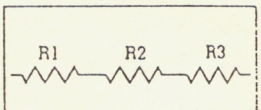
E = I x R
I = E/R
R = E/I

Where
E= Volts
R= Resistance in Ohms
I= Current in Amps

An easier way to remember Ohms Law is by this diagram. Just cover up the value you are looking for, and the formula shows

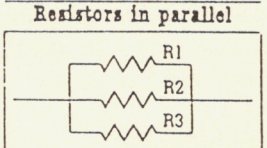


Resistors in series

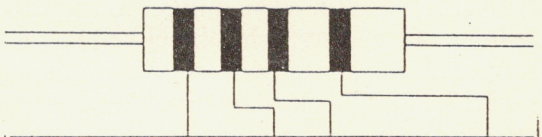


R TOTAL=R1+R2+R3

Resistors in parallel



$$R_{TOTAL} = \frac{1}{\frac{1}{R1} + \frac{1}{R2} + \frac{1}{R3}}$$



COLOUR	TENS	UNITS	MULTIPLIER	TOLERANCE
Black	0	0	1	20% (M)
Brown	1	1	10	1% (F)
Red	2	2	100	2% (G)
Orange	3	3	1000	-
Yellow	4	4	10000	-
Green	5	5	100000	-
Blue	6	6	1000000	-
Violet	7	7	-	-
Grey	8	8	-	-
White	9	9	-	-
Gold	-	-	0.1	5% (J)
Silver	-	-	0.01	10% (K)

Figure D-0.1 Resistor colour codes etc.

DIGITAL LOGIC EXERCISE NO 1.

TOPIC: Digital trainer kit familiarization.

AIM: To understand the working of the various components of the digital kit.

INTRODUCTION:

The central part of the digital kit is the breadboard. The top and the bottom views of the breadboard are shown in figure D-1.0. There are two types of areas on the breadboards. Type A & C is used for making Vcc and GND bus lines. Type B is used for mounting the ICs and other components. The dark thick lines in the bottom view of the breadboard are the metal parts inside it, and therefore indicate the internal connections.

There are a set of four (eight for the internal kit) toggle switches on the lower side of the kit. These can be used for providing input to the circuits. Four push button switches provide debounced outputs. To observe the output of the circuits under test, four (eight for the internal kit) Light Emitting Diodes (LEDs) have been provided. Signal lines for the set of four LEDs are also connected to a 7-segment display through the display driver. It will display the Hexadecimal code of the four bits displayed on the LEDs.

On the external kit the 5 volt DC supply is obtained from a 9 volt battery by using a three terminal regulator. There are four parallel outputs available for 5v as well as Ground (GND). The +12 and the -12 volt supplies on the internal kit will not be used.

The following procedure would also help you to locate any faulty parts of the kit. Inform your tutor about the faults immediately.

PROCEDURE:

1. Switch on the power to the board. The internal kit needs to be plugged into a mains power outlet. In the external kit the 9v battery must be plugged in. Connect a reasonably long wire to one of the LEDs and test the 5v supply by probing it. If the LED turns ON then the power supply is OK.

2. Test all of the LEDs one by one, by connecting to the 5v supply.

3. Connect the output of four toggle switches to the four LEDs, left most switch to the left most LED, etc. Toggle each switch in turn and observe the output on the LEDs. Tabulate all possible combinations of switch settings (up/down), the state of the LEDs (ON/OFF) and the output on the 7-segment display.

4. Test the output of the push button switches. These outputs are normally LO and give a short HI going pulse when the button is pressed. You may not be able to observe this pulse on the LED. If the LED stays ON you may assume, for the time being, that the switch is working OK.

5. Check the continuity of the various connecting strips in the breadboard. Use figure D-1.0 to check out the internal connections. Power up one end of any strip form the 5v line and probe the whole length of the strip with an LED used as a logic probe.

6. Try out any other thing you may like, to feel confidant about the kit's operation. Make sure that you **never connect two outputs together**.

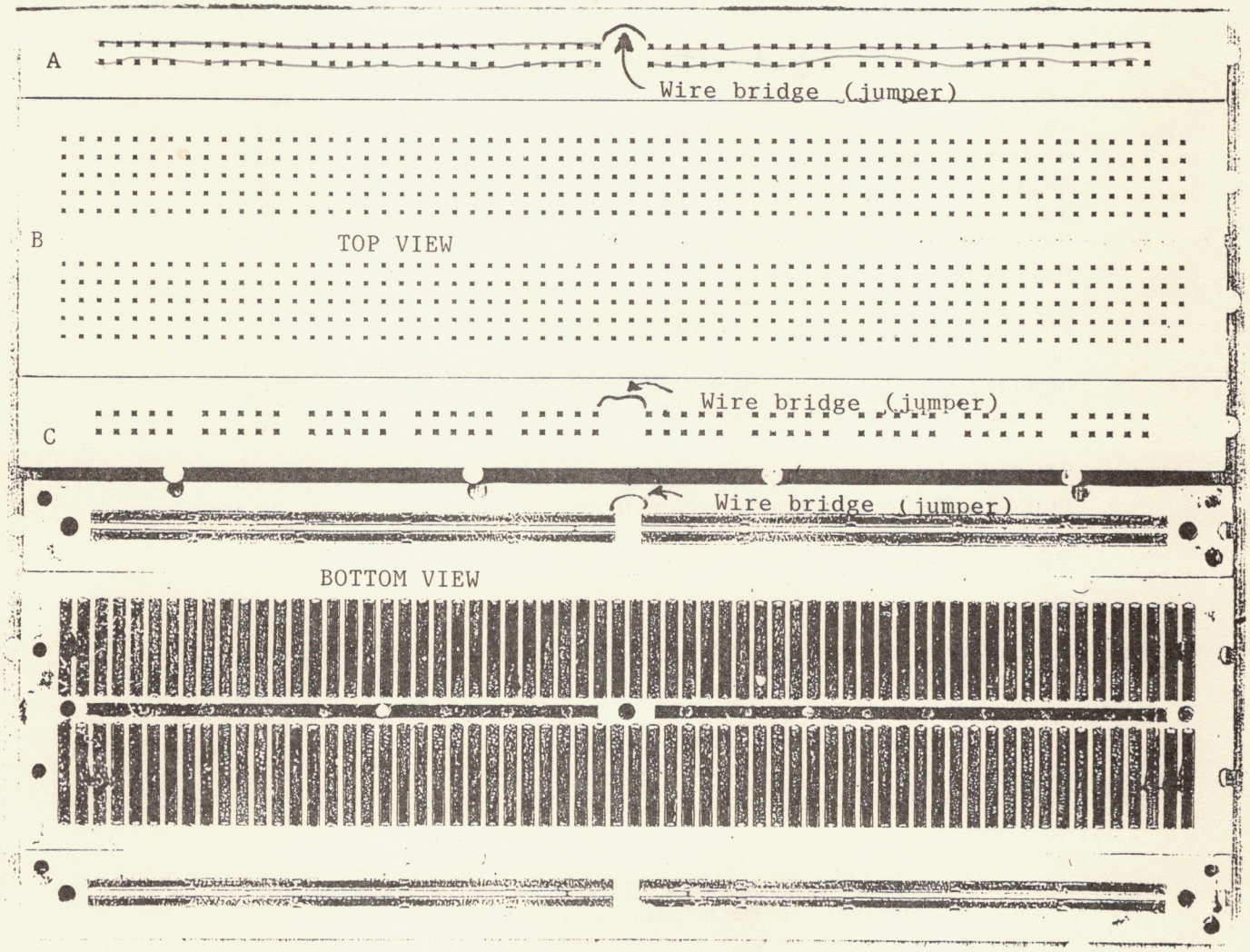


Figure D-1.0 Breadboard top-view and bottom-view.

DIGITAL LOGIC EXERCISE NO 2

TOPIC: Introduction to digital logic integrated circuits.

AIM: To learn about the digital IC families such as TTL and CMOS. To learn practical aspects of wiring digital ICs for making complete digital systems.

To wire a two input to one output multiplexer (2x1Mux) using NAND gates on IC 74LS00. To learn the use of an LED as a logic probe.

INTRODUCTION:

Transistor is the basic component which is used to build most of the electronic systems. Back in the 60s (and early 70s) digital components, such as gates and flipflops were wired on printed circuit boards using transistors and associated components. With the advent of integrated circuits it became possible to fabricate gates, flipflops and now even an entire computer on a single IC chip.

A set of IC chips fabricated by using a specific technology is referred to as as IC family. The two most often used IC families are TTL (transistor transistor logic) and CMOS (complementary metal oxide semiconductor). Two NAND gate fabricated with the two technologies would of course perform the same logic function, but the operating parameters - such as power supply voltage, power dissipation, input current, fan-in fan-out etc. - would be different. The major difference between these two families is that the TTL circuits work off a 5v supply, while the CMOS circuits can work from a supply anywhere between 3v and 15v.

A few 74** series TTL gates are shown in figure D-2.0. Note that pin 14 is Vcc (+5v) for these ICs and pin 7 is GND. This pin numbering is common, but not a universal convention. Pin numbers are identified by a dot on pin number 1 or a notch on the edge adjoining pin number 1. Pins are then counted counter clock-wise, looking from the top.

For further information on digital ICs refer to manufacturer's data books or the following reference.

The Master IC Cookbook by Clayton L. Hallmark, Tab books. Murdoch Library Call No. 621.38173 H191 1.

PROCEDURE:

1. Study the pin numbering and pin assignment for the ICs given in figure D-2.0. Identify the common factors in the system for pin numbering.

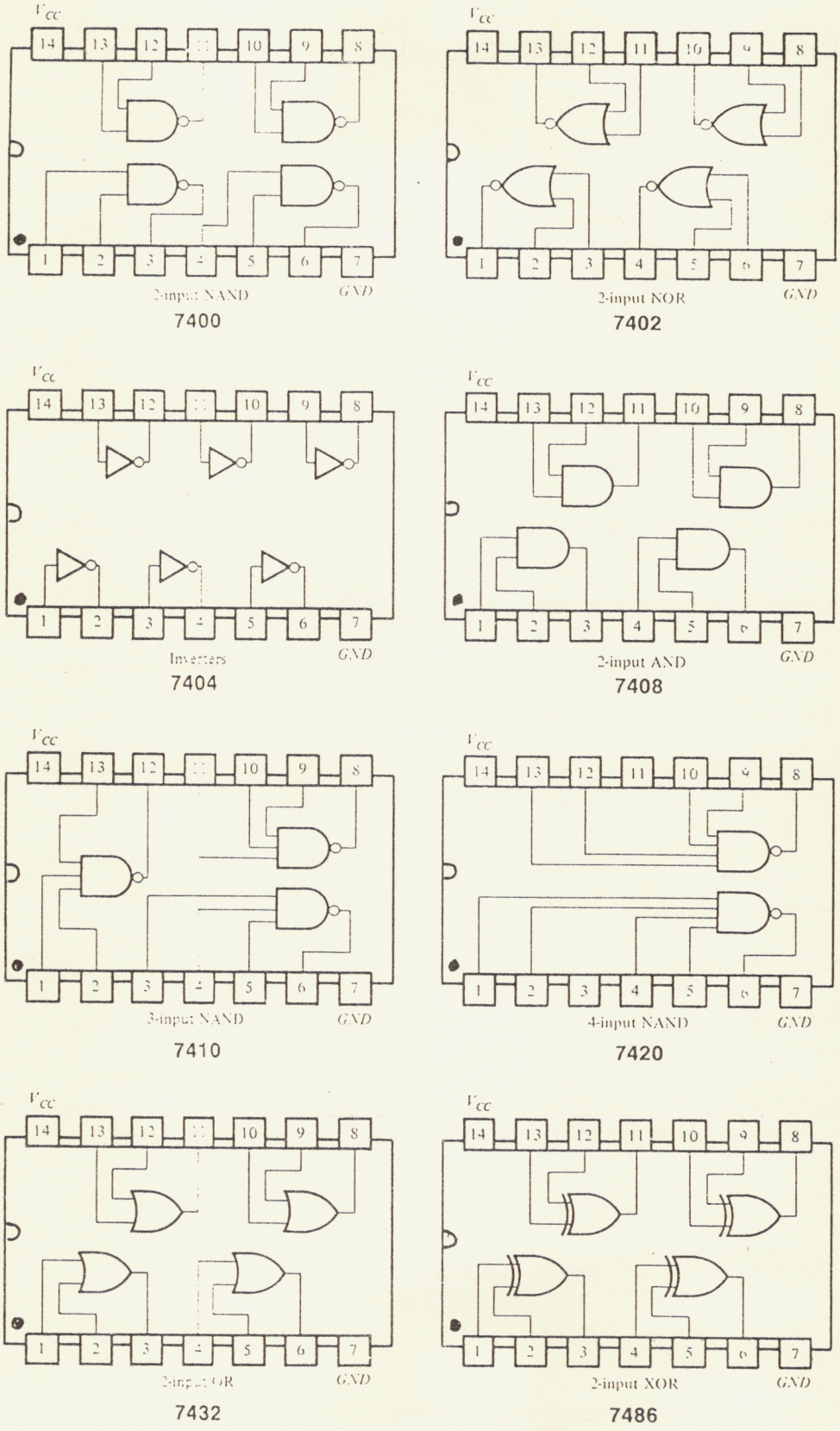


Figure D-2.0 Some digital gates in the 74-- series.**

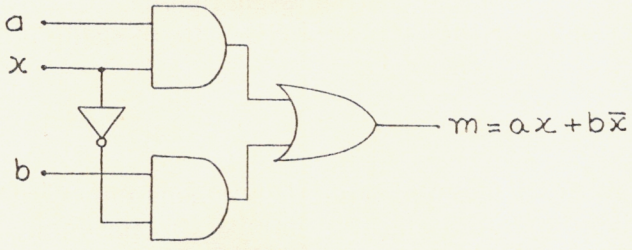
2. The logic diagram for the 2x1 Mux is shown in figure D-2.1a. The output **m** is equal to the input **a** if **x=1**. If **x=0** the **m=b**. The wiring diagram using NAND gates on IC 7400 is shown in figure D-2.1b (74LS00 plugged into your breadboard has the same pin numbering). Confirm that the pin numbers mentioned on this wiring diagram are correct.

   On which pin will the output **m** be available? In further exercises you should write pin numbers on the wiring diagrams before attempting to wire the circuit.

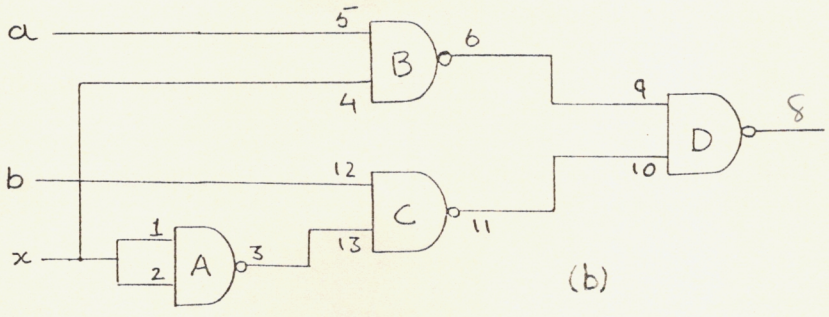
3. Make a Vcc (5v) bus in section A of the breadboard. Connect a short jumper between the two bus sections, as shown in figure D-1.0. Similarly make the GND bus in section C of the breadboard. Connect pin 14 to the Vcc bus and pin 7 to the GND bus.

4. Wire the circuit step by step (i.e. one gate at a time). At each step check the operation of the most recently wired gate by using an LED as a logic probe. The preferred order for wiring the gates would be A-B-C-D. The inputs **a,b** and **x** can be derived from three toggle switches.

5. Test the operation of the complete circuit and record it in a truth table.



(a)



(b)

Figure D-2.1 A two input to one output multiplexer

a) Logic diagram b) Wiring diagram

DIGITAL LOGIC EXERCISE NO 3

TOPIC: Combinational Circuits

AIM: To wire the following combinational circuits and to test their operation.

1. An Exclusive-OR using NAND gates.
2. A 2 to 4 line decoder using NAND and NOR gates.

INTRODUCTION:

The NAND gate is the basic gate that can be fabricated on an integrated circuit. Also, any combinational logic equation can be implemented by using only two input NAND gates. The exclusive-OR function of two boolean variable is true if the two variables are not equal to each other. This function can be implemented by using two input NAND gates, as shown in figure D-3.0. Note that an OR gate with inputs inverted is nothing but a NAND gate; by DeMorgan's theorem.

A two line to four line decoder is a circuit in which one of the four output lines is true depending upon the code set on the two input lines. The logic diagram of this circuit is shown in figure D-3.1a. This circuit can be implemented as shown in figure D-3.1b, by using NAND and NOR gates.

PROCEDURE:

1. Redraw the diagram of figure D-3.0 using only NAND gates. On this diagram assign pin numbers to the inputs and the outputs of the gates using the 74LS00 chip.

2. Wire up the circuit as per your wiring diagram. Use the technique of step by step wiring and checking. Record the function of the circuit in a truth-table, including the intermediate outputs of the gates C,D&E.

3. Draw the wiring diagram for the 2-to-4 decoder, i.e. assign pin numbers to the gates in figure D-3.1b.

4. Wire and test the operation of the circuit. Tabulate the output in a truth-table.

$$X(\overline{X} + \overline{Y}) + Y(\overline{X} + \overline{Y}) = X\overline{Y} + \overline{X}Y$$

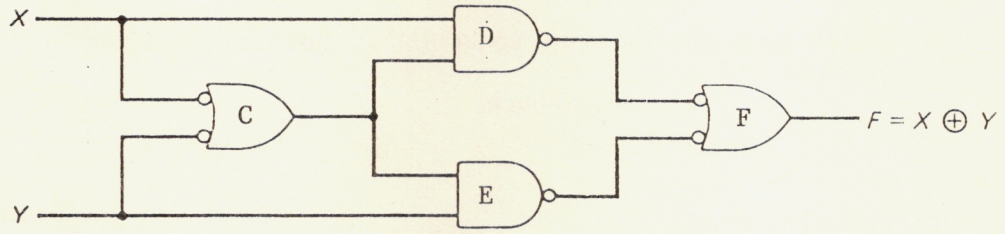


Figure D-3.0 **Exclusive-OR constructed with NAND gates.**

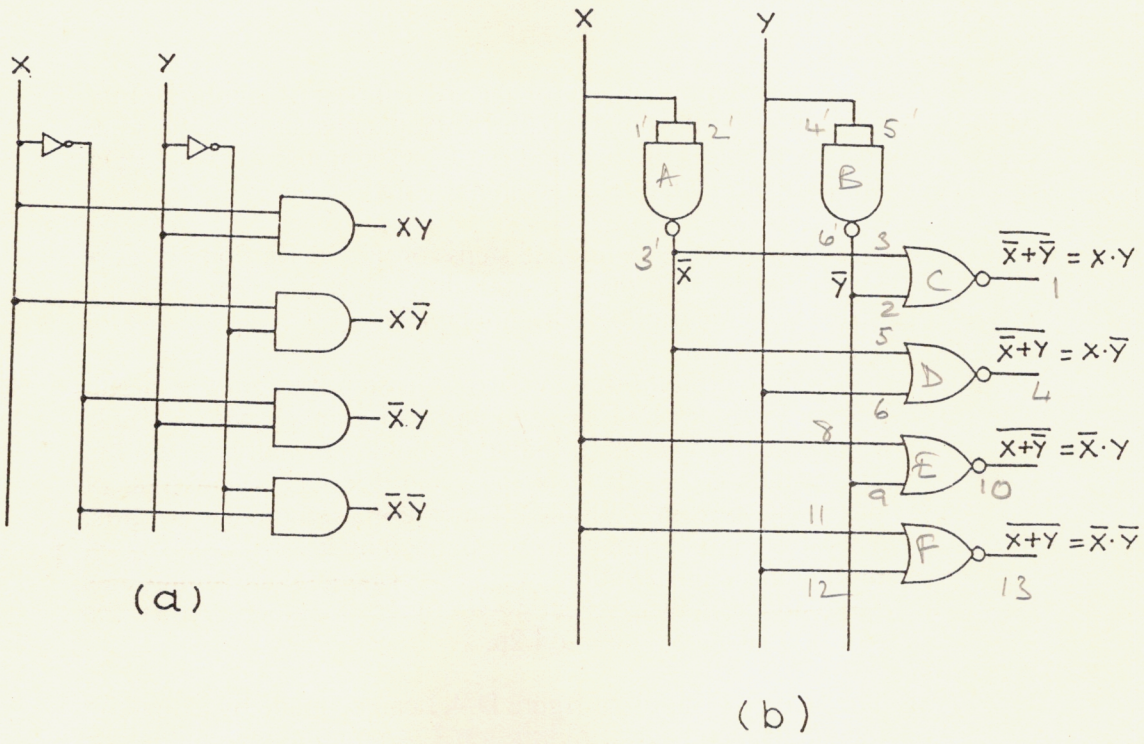


Figure D-3.1 **Two line to four line decoder**
a) Logic diagram b) Wiring diagram

DIGITAL LOGIC EXERCISE NO 4

TOPIC: Sequential Circuits

AIM: To wire the following sequential circuits and to test their operation.

1. SR flipflop and latch.
2. D-type latch.

INTRODUCTION:

The SR flipflop is the building block for most of the sequential circuits. An SR flipflop can be built by using NOR gates (figure D-4.0) as well as with NAND gates (figure D-4.1). An SR latch is formed by adding a control input to the basic SR flipflop, as shown in figure D-4.2.

The undefined states of an SR latch can be avoided if the R input is generated by inverting the S input, as shown in figure D-4.3. This is the D-type latch.

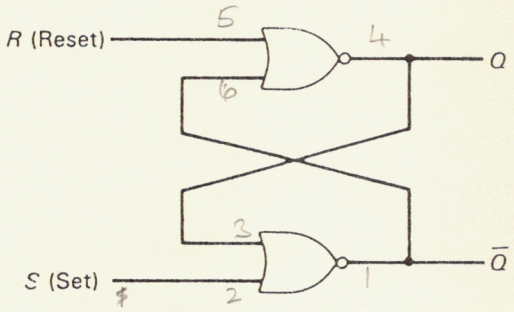
PROCEDURE:

1. Draw a wiring diagram for the SR-flipflop of figure D-4.0a. Wire the circuit using the 74LS02 chip. connect the S, R inputs to two toggle switches and the Q, Q' outputs to two LEDs. Verify the function table given in figure D-4.0b.

2. Repeat the above mentioned steps for the SR flipflop of figure D-4.1a and using the 74LS00 chip. How does this flipflop differ from the flipflop of figure D-4.0?

3. Extend the circuit you have wired in step 2 to get the circuit shown in figure D-4.2a. You can still manage with a single 74LS00 chip. The control input C should normally remain LO and be pulsed HI to latch-in the S and R inputs. ~~The outputs of the push button switches work complementary to this.~~ Therefore, the C input can be obtained ~~by inverting the output~~ from the outputs of one of the push button switches. ~~Use one of the NOR gates as an inverter by connecting its inputs together~~. Verify the function table of figure D-4.2b.

4. The D-type Flipflop of figure D-4.3 can be made by extending the circuit you have wired in step 3. Again, the inverter can be made by connecting together the two inputs of a NOR gate from the 74LS02 chip. Confirm that the input data (D input) is latched when a clock pulse is applied.

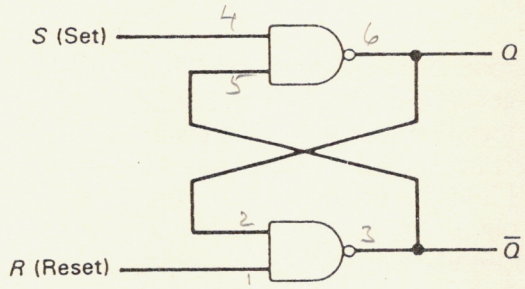


(a) Logic diagram

S	R	Q	$\bar{Q}$	
1	0	1	0	Set state
0	0	1	0	
0	1	0	1	Reset state
0	0	0	1	
1	1	0	0	Undefined

(b) Function table

Figure D-4.0 SR Flipflop with NOR gates

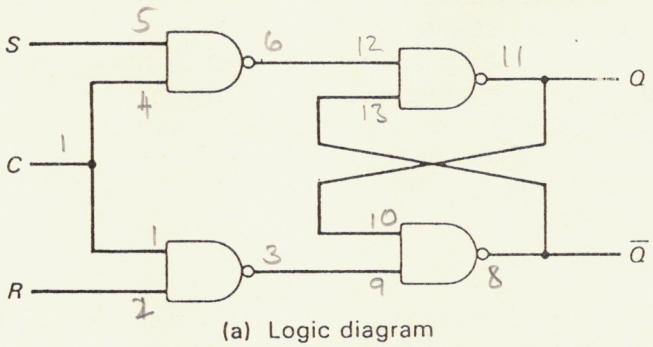


(a) Logic diagram

S	R	Q	$\bar{Q}$	
0	1	1	0	Set state
1	1	1	0	
1	0	0	1	Reset state
1	1	0	1	
0	0	1	1	Undefined

(b) Function table

Figure D-4.1 SR Flipflop with NAND gates

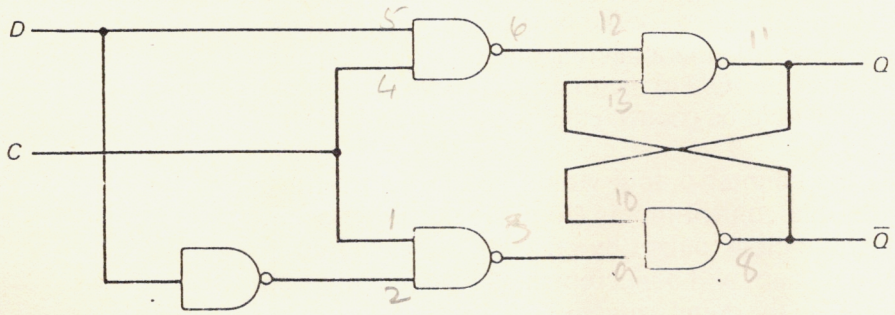


(a) Logic diagram

C	S	R	Next state of Q
0	X	X	No change
1	0	0	No change
1	0	1	Q = 0; Reset state
1	1	0	Q = 1; Set state
1	1	1	Indeterminate

(b) Function table

Figure D-4.2 SR Latch with control input



(a) Logic diagram

C	D	Next state of Q
0	X	No change
1	0	Q = 0; Reset state
1	1	Q = 1; Set state

(b) Function table

Figure D-4.3 D-type Latch

DIGITAL LOGIC EXERCISE NO 5

TOPIC: 4-Bit Binary Counter

AIM: To study IC 74161, 4-bit synchronous binary counter. To construct a modulo-6 counter using this chip.

INTRODUCTION:

The 74161 is a 4-bit synchronous counter with parallel load and asynchronous clear. The internal logic of this IC is similar to that shown in figure 5.18 of the text book by Mano. Its pin assignment and function table are given in figure D-5.0.

This circuit has four parallel data inputs, one clock input (rising edge triggered), and three control inputs. The control inputs are Load, Count and Clear. From the function table it can be seen that if the Clear input is pulled LO the outputs are cleared irrespective of the state of any other input. With the Clear input HI and the Load input LO the data on the input lines is loaded onto the output lines at the occurrence of a clock pulse. When all three control inputs, i.e. Clear, Load and Count, are HI the counter is incremented on the positive edge of the clock pulse. The upward arrow in the function table indicates that the action takes place on the rising (positive) edge of the clock pulse.

A modulo-n counter counts in the range 0 to n-1. Figure D-5.1 shows four ways in which a modulo-6 counter can be obtained from a 4-bit binary counter. In general, a modulo-n counter can be constructed from a p-bit binary counter if 2 ** p >= n . What is the smallest binary counter that can be used to make a modulo-6 counter? Draw the block diagram of a mod-6 counter using the smallest possible binary counter.

PROCEDURE:

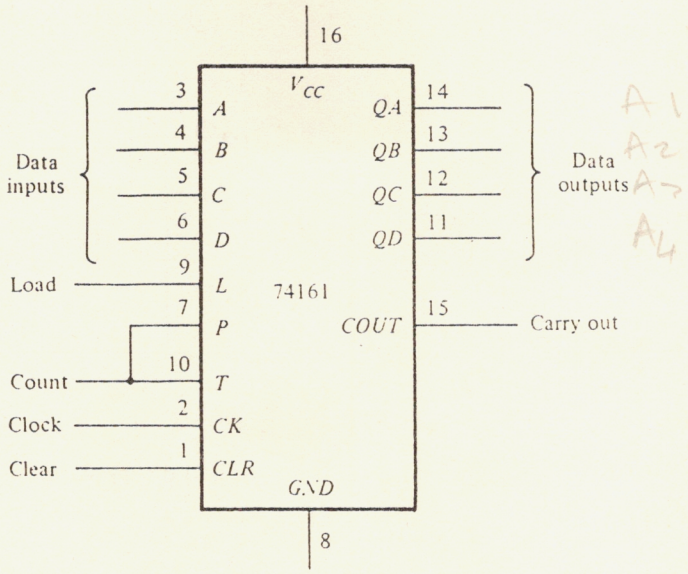
1. Wire up the 74LS161. Connect the control inputs Load, Clear and the two Count inputs (pin 7 & 10) to the four toggle switches. The Count input as shown in the function table is the logical AND of the two Count input lines. Therefore both of these must be high for the counting to proceed. Connect the Clock input to one of the bush-button switches. The four data inputs can be left unconnected for the time being. Note that the TTL circuits assume all unconnected inputs as logical HI. Connect the outputs QA - QD to the four LEDs. QD is the most significant bit and should be connected to the left-most LED.

2. To begin with, the counter output would be at any arbitrary value. Keep all three control inputs (four toggle switches) at logical HI. Generate clock pulses by pressing the push button switch. The output count will be incremented by one with each clock pulse. Tabulate the count sequence.

3. Make the Clear input LO. This should reset all the output bits to LO. Set the Clear input back to HI, make the Load input LO and apply a clock pulse. This should make all the outputs HI. This confirms that the unconnected input lines are taken as logical HI.

4. Confirm that both of the Count inputs have to be logical HI for the counting to proceed.

5. Wire the mod-6 counters shown in figure D-5.1, one at a time. The four input data lines can be set to HI by connecting it to the Vcc (5v) bus through a pull-up resistor (see footnote 1). The pull-up resistor can be any value between 1 K ohms and 10 K ohms. Inputs lines you wish to set to logical LO can be connected to the GND bus. The extra set of four toggle switches can be used on the internal kits. In the circuit of figure D-5.1a you require an AND gate, and there is none supplied. How would you manage with the available chips? Tabulate the output of each of these circuits.

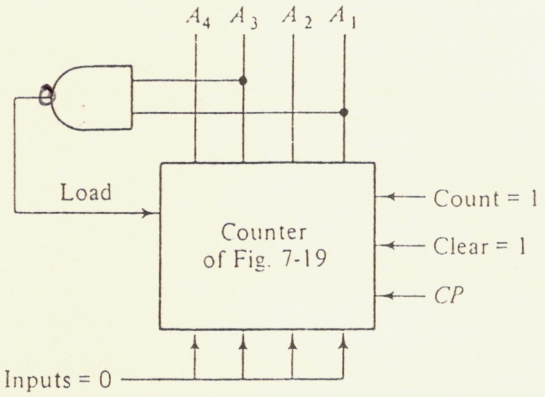


Function table

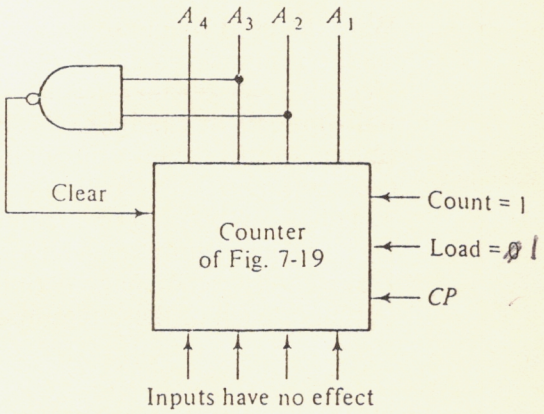
Clear	Clock	Load	Count	Function
0	X	X	X	Clear outputs to 0
1	↑	0	X	Load input data
1	↑	1	1	Count to next binary value
1	↑	1	0	No change in output

Figure D-5.0 4-bit binary counter, IC 74161.

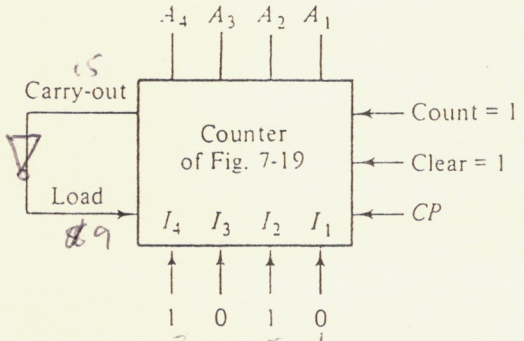
Footnote 1: To pull up an input to 5volts connect one end of the pull-up resistor to the input pin and the other end of the resistor to the Vcc bus.



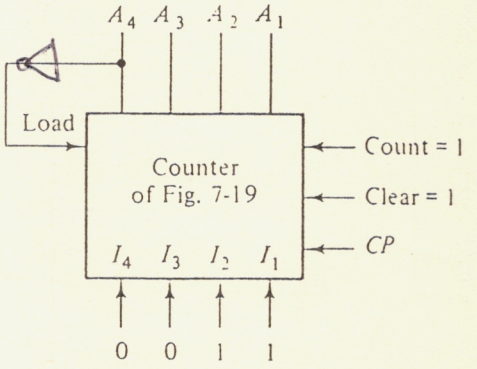
(a) Binary states 0, 1, 2, 3, 4, 5.



(b) Binary states 0, 1, 2, 3, 4, 5.



(c) Binary states 10, 11, 12, 13, 14, 15.



(d) Binary states 3, 4, 5, 6, 7, 8.

Figure D-5.1 **Four circuits for making a modulo-6 counter using a 4-bit binary counter.**

DIGITAL LOGIC DESIGN EXERCISE

TOPIC: Design with JK flipflops.

AIM: Design a sequential circuit with two JK flipflops A and B and two inputs E and X. If E = 0, the circuit remains in the same state regardless of the value of X. When E = 1 and X = 1, the circuit goes through the state transitions 00-01-10-11-00 and repeats the sequence. When E = 1 and X = 0, the circuit goes through the state transitions 00-11-10-01-00 and repeats the sequence.

INTRODUCTION:

This is an Up-Down counter with an Enable input. The E input being the enable signal and the X input determines if the counter will count up or down. The design procedure that you need to use is given in section 4-6 of the text by Mano. In particular you should understand the JK flipflop excitation table given in Table 4-6, on page 145 of the text.

PROCEDURE:

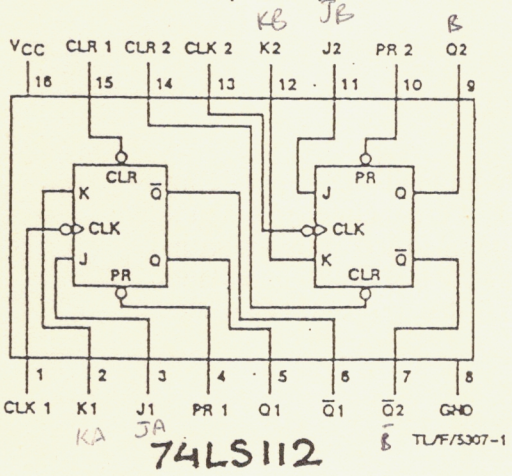
1. Make a state table for the given problem, similar to that given in table 4-7 of the text. By using Karnaugh maps derive the equations for the flipflop inputs JA, KA, JB, and KB. Make full use of the don't care conditions. Draw a logic diagram for the circuit.

2. You have to implement the design using the dual JK flipflop chip 74LS112. The chip pin numbers and the function table are given in figure D-6.0. Try out the function of the JK flipflops before you wire up the circuit.

3. Since you have only NAND and NOR gates available on the kit, transform your equations to use these gates. Draw a wiring diagram of the circuit with pin numbers indicated on the inputs and the outputs of all the components in the diagram.

4. Wire up the circuit using two toggle switches as the inputs and two LEDs as the outputs. Use one of the push button switches as the clock input. External students should label the inputs switches so that the tutor can check the operation of the circuit easily.

Dual J-K Flip-Flop with Preset and Clear



Truth Table

Inputs					Outputs	
PR	CLR	CLK	J	K	Q	$\overline{Q}$
L	H	X	X	X	H	L
H	L	X	X	X	L	H
L	L	X	X	X	L*	L*
H	H	↓	L	L	Q_0	$\overline{Q}_0$
H	H	↓	H	L	H	L
H	H	↓	L	H	L	H
H	H	↓	H	H	TOGGLE	
H	H	H	X	X	Q_0	$\overline{Q}_0$

*This is an unstable condition, and is not guaranteed

Figure D-6.0 Dual JK flipflop, 74LS112.

Copied pursuant to S53B of
the Copyright Act as amended
on behalf of Murdoch Uni.
on 17.8.89

Reprinted from

Bartee, Thomas C. Digital computer fundamentals
6th ed McGraw-Hill 1985 pp xii + 610 ISBN 0 07 0038996

The following reprints are questions from selected chapters, together with the answers to selected odd-numbered questions.

51 - 56 Number systems
119 - 124 Boolean algebra and gate networks
186 - 188 Logic design
253 - 257 The arithmetic-logic unit
318 - 319 The memory element
584 - 602 Answers to selected questions

2

QUESTIONS

QUESTIONS

2.1 Convert the following decimal numbers to equivalent binary numbers:

(*a*) 43 (*b*) 64 (*c*) 4096
(*d*) 0.375 (*e*) $\frac{27}{32}$ (*f*) 0.4375
(*g*) 512.5 (*h*) 131.5625 (*i*) 2048.0625

2.2 Convert the following numbers to the equivalent binary numbers:

(*a*) 14 (*b*) 0.25 (*c*) $2\frac{1}{8}$
(*d*) 6.25 (*e*) $2\frac{3}{8}$ (*f*) 0.625

2.3 Convert the following binary numbers to equivalent decimal numbers:

(*a*) 1101 (*b*) 11011 (*c*) 1011
(*d*) 0.1011 (*e*) 0.001101 (*f*) 0.001101101
(*g*) 111011.1011 (*h*) 1011011.001101 (*i*) 10110.0101011101

2.4 Convert the following binary numbers to equivalent decimal numbers:

(*a*) 1011 (*b*) 11000 (*c*) 100011
(*d*) 11011 (*e*) 111001 (*f*) 1011010

2.5 Convert the following binary numbers to equivalent decimal numbers:

(*a*) 1011 (*b*) 100100 (*c*) 10011
(*d*) 0.1101 (*e*) 0.1001 (*f*) 0.0101
(*g*) 1011.0011 (*h*) 1001.1001 (*i*) 101.011

2.6 Convert the following binary numbers to equivalent decimal numbers:

(*a*) 0.111 (*b*) 0.11011 (*c*) 1.011
(*d*) 111.1011 (*e*) 0110.0101 (*f*) 101.101011

2.7 Perform the following additions and check by converting the binary numbers to decimal:

(*a*) 1001.1 + 1011.01 (*b*) 100101 + 100101
(*c*) 0.1011 + 0.1101 (*d*) 1011.01 + 1001.11

2.8 Perform the following additions and check by converting the binary numbers to decimal and adding:

(*a*) 1011 + 1110 (*b*) 1010 + 1111 (*c*) 10.11 + 10.011
(*d*) 1101.11 + 1.11 (*e*) 11111.1 + 10010.1 (*f*) 101.1 + 111.11

2.9 Perform the following additions and check by converting the binary numbers to decimal:

(*a*) 1101.1 + 1011.1 (*b*) 101101 + 1101101
(*c*) 0.0011 + 0.1110 (*d*) 1100.011 + 1011.011

2.10 Perform the following subtractions in binary and check by converting the numbers to decimal and subtracting:

(*a*) 1101 − 1000 (*b*) 1101 − 1001 (*c*) 1011.1 − 101.1
(*d*) 1101.01 − 1011.1 (*e*) 111.11 − 101.1 (*f*) 1101.1 − 1010.01

2.11 Perform the following subtractions in the binary number system:

(*a*) 64 − 32 (*b*) 127 − 63
(*c*) 93.5 − 42.75 (*d*) $84\frac{9}{32} - 48\frac{5}{16}$

2.12 Perform the following subtractions in the binary number system:

(*a*) 128 − 32 (*b*) $\frac{1}{8} - \frac{1}{16}$ (*c*) $2\frac{1}{8} - 4\frac{3}{32}$
(*d*) $31 - \frac{5}{8}$ (*e*) $62 - 31\frac{1}{16}$ (*f*) 129 − 35

2.13 Perform the following subtractions in the binary number system:

(*a*) 37 − 35 (*b*) 128 − 64
(*c*) 94.5 − 43.75 (*d*) 255 − 127

2.14 Perform the following multiplications and divisions in the binary number system:

(*a*) 16 × 8 (*b*) 31 × 14 (*c*) 23 × 3.525
(*d*) 15 × 8.625 (*e*) 6 ÷ 2 (*f*) 16 ÷ 8

2.15 Perform the following multiplications and divisions in the binary number system:

(*a*) 24 × 12 (*b*) 18 × 14 (*c*) 32 ÷ 8
(*d*) 27 ÷ 18 (*e*) 49.5 × 51.75 (*f*) 58.75 ÷ 23.5

2.16 Perform the following multiplications and divisions in the binary number system:

(*a*) 16 × 2.75 (*b*) 19 ÷ 6 (*c*) $256\frac{1}{2} \div 128\frac{1}{4}$
(*d*) 31.5 ÷ 15.75 (*e*) $3 \div \frac{5}{8}$ (*f*) $2\frac{5}{8} \times 1\frac{5}{8}$

2.17 Perform the following multiplications and divisions in the binary number system:

(*a*) 15 × 13 (*b*) 10 × 15 (*c*) 44 ÷ 11
(*d*) 42 ÷ 12 (*e*) 7.75 × 2.5 (*f*) 22.5 × 4.75

QUESTIONS

2.18 Convert the following decimal numbers to both their 9s and 10s complements:

(*a*) 9 (*b*) 19 (*c*) 8
(*d*) 24 (*e*) 25 (*f*) 99

2.19 Convert the following decimal numbers into both their 9s and 10s complements:

(*a*) 5436 (*b*) 1932 (*c*) 45.15 (*d*) 18.293

2.20 Convert the following decimal numbers into both their 9s and 10s complements:

(*a*) 95 (*b*) 79 (*c*) 0.83
(*d*) 0.16 (*e*) 298.64 (*f*) 332.52

2.21 Convert the following decimal numbers to both their 9s and 10s complements:

(*a*) 3654 (*b*) 2122 (*c*) 54.19 (*d*) 37.263

2.22 Convert the following binary numbers to both their 1s and 2s complements:

(*a*) 1101 (*b*) 1010 (*c*) 1111
(*d*) 1110 (*e*) 1011 (*f*) 1011

2.23 Convert the following binary numbers to both their 1s and 2s complements:

(*a*) 1011 (*b*) 11011 (*c*) 1011.01 (*d*) 11011.01

2.24 Convert the following binary numbers to both their 1s and 2s complements:

(*a*) 1011 (*b*) 1101 (*c*) 0.0111
(*d*) 0.101 (*e*) 11.101 (*f*) 101.011

2.25 Convert the following binary numbers to both their 1s and 2s complements:

(*a*) 101111 (*b*) 100100
(*c*) 10111.10 (*d*) 10011.11

2.26 Perform the following subtractions, using both 9s and 10s complements:

(*a*) 8 − 4 (*b*) 16 − 8 (*c*) 198 − 124
(*d*) 28.5 − 23.4 (*e*) 27.6 − 23.4 (*f*) 0.55 − 0.42

2.27 Perform the following subtractions, using both 9s and 10s complements:

(*a*) 948 − 234 (*b*) 347 − 263
(*c*) 349.5 − 245.3 (*d*) 412.7 − 409.2

2.28 Perform the following subtractions, using both 9s and 10s complements:

(*a*) 14 − 9 (*b*) 15 − 9 (*c*) 0.5 − 40.24
(*d*) 0.41 − 0.4 (*e*) 0.434 − 0.33 (*f*) 1.2 − 0.34

2.29 Perform the following subtractions, using both 9s and 10s complements:

(*a*) 1024 − 913 (*b*) 249 − 137
(*c*) 24.1 − 13.4 (*d*) 239.3 − 119.4

2.30 Perform the following subtractions of binary numbers, using both 1s and 2s complements:

(*a*) 1010 − 1011 (*b*) 110 − 10 (*c*) 110 − 0.111
(*d*) 0.111 − 0.1001 (*e*) 0.1111 − 0.101 (*f*) 11.11 − 10.111

QUESTIONS

2.46 Convert the following octal numbers to decimal:
(*a*) 15 (*b*) 125 (*c*) 115
(*d*) 124 (*e*) 156 (*f*) 15.6

2.47 Convert the following octal numbers to decimal:
(*a*) 2376 (*b*) 2473 (*c*) 276431 (*d*) 22632

2.48 Convert the following binary numbers to octal:
(*a*) 110 (*b*) 111001 (*c*) 111.111
(*d*) 0.11111 (*e*) 10.11 (*f*) 1111.1101

2.49 Convert the following binary numbers to octal:
(*a*) 101101 (*b*) 101101110 (*c*) 10110111
(*d*) 110110.011 (*e*) 011.1011011

2.50 Convert the following octal numbers to binary:
(*a*) 54 (*b*) 44 (*c*) 232.2
(*d*) 232.4 (*e*) 453.45 (*f*) 31.234

2.51 Convert the following octal numbers to binary:
(*a*) 7423 (*b*) 3364 (*c*) 33762
(*d*) 3232.14 (*e*) 3146.52

2.52 Convert the following decimal numbers to octal:
(*a*) 17 (*b*) 8 (*c*) 19
(*d*) 0.55 (*e*) 0.625 (*f*) 2.125

2.53 Convert the following decimal numbers to octal:
(*a*) 932 (*b*) 332 (*c*) 545.375
(*d*) 632.97 (*e*) 4429.625

2.54 Convert the following hexadecimal numbers to binary:
(*a*) 9 (*b*) 1B (*c*) 0.A1
(*d*) 0.AB (*e*) AB (*f*) 12.B

2.55 Convert the following hexadecimal numbers to binary:
(*a*) CD (*b*) 6A9 (*c*) A14
(*d*) AA.1A (*e*) AB2.234

2.56 Convert the following binary numbers to hexadecimal:
(*a*) 1101.0110 (*b*) 11011110 (*c*) 1111
(*d*) 11101 (*e*) 11110.01011 (*f*) 1011.11010

2.57 Convert the following binary numbers to hexadecimal:
(*a*) 10110111 (*b*) 10011100 (*c*) 1001111
(*d*) 0.01111110 (*e*) 101101111010

2.58 A simple rule for multiplying two digits in any radix is simply to multiply the two digits in decimal. If the product is less than the radix, take it; if greater, divide (in decimal) by the radix and use the remainder as the first, or least significant, position and the quotient as the carry, or most significant, digit. In base 6, then $2 \times 2 = 4$, $3 \times 1 = 3$, etc.; however, $2 \times 4 = 8$, and

$$6\overline{)8}\,\begin{matrix}1\end{matrix}$$

So $2_6 \times 4_6 = 12_6$. Similarly, in base 7, then, $3 \times 4 = 12$ and

$$7\overline{)12}\ \ 1 \qquad \underline{7} \qquad 5$$

So $3_7 \times 4_7 = 15_7$. Using this rule, perform, in base 7, the following:

(*a*) $2_7 \times 3_7$ (*b*) $2_7 \times 2_7$
(*c*) $4_7 \times 4_7$ (*d*) $4_7 \times 3_7$

2.59 Using the rule in Question 2.58, perform:

(*a*) $3_6 \times 4_6$ (*b*) $3_6 \times 3_6$ (*c*) $3_9 \times 4_9$
(*d*) $4_9 \times 5_9$ (*e*) $5_9 \times 15_9$

2.60 An addition table for octal is as follows:

	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7	10
2	2	3	4	5	6	7	10	11
3	3	4	5	6	7	10	11	12
4	4	5	6	7	10	11	12	13
5	5	6	7	10	11	12	13	14
6	6	7	10	11	12	13	14	15
7	7	10	11	12	13	14	15	16

Using this table, we add in octal:

$$\begin{array}{r} 1\ 1 \longleftarrow \text{carries} \\ 1\ 2\ 6 \\ 3\ 5\ 7 \\ \hline 5\ 0\ 5 \end{array}$$

Perform the following additions:

(*a*) $7_8 + 7_8$ (*b*) $6_8 + 5_8$ (*c*) $7_8 + 16_8$
(*d*) $5_8 + 4_8$ (*e*) $5_8 + 14_8$

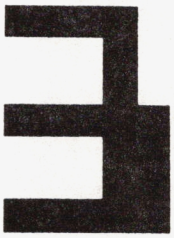
2.61 Using the table in Question 2.60, perform:

(*a*) $15_8 + 14_8$ (*b*) $24_8 + 36_8$ (*c*) $126_8 + 347_8$
(*d*) $67_8 + 45_8$ (*e*) $136_8 + 636_8$

2.62 Make up a hexadecimal addition table.

2.63 Using the table in Question 2.62, perform:

(*a*) $6_{16} + A1_{16}$ (*b*) $7_{16} + 17_{16}$ (*c*) $8_{16} + 28_{16}$
(*d*) $A16A_{16} + B16A_{16}$ (*e*) $A84_{16} + A83_{16}$



This part of the IC business is sufficiently developed that PLA designs can be punched into cards and sent to a manufacturer, punched into tapes and sent to a manufacturer, and even sent over the telephone line from a terminal connected to a tape reader. (Manufacturers will provide a long-distance number you can call to phone in designs.)

SUMMARY

3.27 This chapter presented the basic idea of a gate and showed how gates can be interconnected to form logic networks. The basic types of gates were introduced: AND, OR, NAND, and NOR gates and inverters.

The design of logic networks for computers is greatly facilitated by boolean algebra. This subject was introduced, and tables of combination, theorems of the algebra, and algebraic reduction of expressions were all explained.

The map method for simplifying boolean algebra expressions was presented. This makes it possible to minimize boolean expressions, thereby also simplifying the logic networks that realize these expressions. The various network forms such as AND-to-OR, NAND-to-NAND, OR-to-AND, NOR-to-NOR, NAND-to-AND, and NOR-to-OR networks were explained, and the design procedure for each was presented.

Gating networks can now be made with many gates in a single container by using LSI manufacturing techniques. The general layout for several such integrated circuits was presented, as was a special representation which is commonly used. This makes possible design of relatively large arrays of gates using PLAs, PALs, and HALs.

QUESTIONS

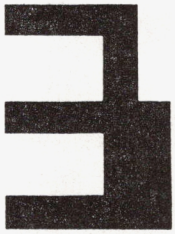
3.1 Prepare a truth table for the following boolean expressions:
(*a*) $XYZ + \overline{X}\overline{Y}\overline{Z}$ (*b*) $ABC + A\overline{B}\overline{C} + \overline{A}\overline{B}\overline{C}$
(*c*) $A(B\overline{C} + \overline{B}C)$ (*d*) $(A + B)(A + C)(\overline{A} + \overline{B})$

3.2 Prepare a table of combinations for the following boolean algebra expressions:
(*a*) $\overline{X}\overline{Y} + \overline{X}Y$ (*b*) $XY\overline{Z} + \overline{X}\overline{Y}Z$ (*c*) $\overline{X}YZ + X\overline{Y}$
(*d*) $\overline{X}\overline{Y}\overline{Z} + X\overline{Y}\overline{Z} + \overline{X}Y\overline{Z}$ (*e*) $\overline{X}\overline{Y} + \overline{Y}\overline{Z}$ (*f*) $\overline{A}B(\overline{A}\overline{B}\overline{C} + \overline{B}C)$

3.3 Prepare a truth table for the following boolean expressions:
(*a*) $A\overline{B} + \overline{A}B$ (*b*) $A\overline{B} + B\overline{C}$
(*c*) $A\overline{C} + AC$ (*d*) $\overline{A}\overline{B}C + AB\overline{C} + \overline{A}BC$
(*e*) $AB(A\overline{B}C + A\overline{B}\overline{C} + AB\overline{C})$

3.4 Prepare a table of combinations for the following boolean algebra expressions:
(*a*) $X(\overline{Y} + \overline{Z}) + X\overline{Y}$ (*b*) $X\overline{Y}(Z + Y\overline{Z}) + \overline{Z}$
(*c*) $[X(Y + \overline{Y}) + \overline{X}(\overline{Y} + Y)]\cdot\overline{Z}$ (*d*) $AB(\overline{A}B + A\overline{B})$
(*e*) $A[(\overline{B} + C) + \overline{C}]$ (*f*) $\overline{A}\overline{B}\overline{C}(\overline{A}B\overline{C} + A\overline{B}\overline{C})$

3.5 Prepare a table of combinations for the following boolean algebra expressions:
(*a*) $XY + \overline{X}\overline{Y}Z$ (*b*) $ABC + A\overline{B} + \overline{A}B$
(*c*) $ABC + \overline{A}\overline{C}$



3.6 Prepare a table of combinations for the following boolean algebra expressions:
(*a*) $A\bar{B}\bar{C} + \bar{A}B$ (*b*) $\bar{A}\bar{B}\bar{C} + AC + AB$
(*c*) $XZ + X\bar{Y} + \bar{X}\bar{Z}$

3.7 Simplify the following expressions, and draw a block diagram of the circuit for each simplified expression, using AND and OR gates. Assume the inputs are from flip-flops.
(*a*) $A\bar{B}\bar{C} + \bar{A}\bar{B}\bar{C} + \bar{A}B\bar{C} + \bar{A}\bar{B}C$
(*b*) $ABC + \bar{A}BC + A\bar{B}C + AB\bar{C} + A\bar{B}\bar{C} + \bar{A}B\bar{C} + \bar{A}\bar{B}\bar{C}$
(*c*) $A(A + B + C)(\bar{A} + B + C)(A + \bar{B} + C)(A + B + \bar{C})$
(*d*) $(A + B + C)(A + \bar{B} + \bar{C})(A + B + \bar{C})(A + \bar{B} + C)$

3.8 Simplify the expressions in Question 3.4 and draw block diagrams of gating networks for your simplified expressions, using AND gates, OR gates, and inverters.

3.9 Simplify the following expressions:
(*a*) $ABC(AB\bar{C} + A\bar{B}C + \bar{A}BC)$ (*b*) $AB + A\bar{B} + \bar{A}C + \bar{A}\bar{C}$
(*c*) $XY + XY\bar{Z} + XY\bar{Z} + XZY$ (*d*) $XY(\bar{X}Y\bar{Z} + X\bar{Y}\bar{Z} + \bar{X}\bar{Y}\bar{Z})$

3.10 Simplify the expressions in Question 3.6 and draw block diagrams of gating networks for your simplified expressions, using AND gates, OR gates, and inverters.

3.11 Form the complements of the following expressions. For instance, the complement of $(XY + XZ)$ is equal to $\overline{(XY + XZ)} = (\bar{X} + \bar{Y})(\bar{X} + \bar{Z}) = \bar{X} + \bar{Y}\bar{Z}$.
(*a*) $(A + BC + AB)$ (*b*) $(A + B)(B + C)(A + C)$
(*c*) $AB + \bar{B}C + CD$ (*d*) $AB(\bar{C}D + \bar{B}C)$
(*e*) $A(B + C)(\bar{C} + \bar{D})$

3.12 Complement the following expressions (as in Question 3.11):
(*a*) $\bar{X}\bar{Y} + X\bar{Y}$ (*b*) $X\bar{Y}Z + \bar{X}Y$
(*c*) $\bar{X}(Y + \bar{Z})$ (*d*) $X(Y\bar{Z} + \bar{Y}Z)$
(*e*) $XY(\bar{Y}Z + X\bar{Z})$ (*f*) $XY + \bar{X}\bar{Y}(Y\bar{Z} + \bar{X}\bar{Y})$

3.13 Prove the two basic De Morgan theorems, using the proof by perfect induction.

3.14 Prove the following rules using the proof by perfect induction:
(*a*) $X\bar{Y} + XY = X$
(*b*) $X + \bar{X}Y = X + Y$

3.15 Convert the following expressions to sum-of-products form:
(*a*) $(A + B)(\bar{B} + C)(\bar{A} + C)$
(*b*) $(\bar{A} + C)(\bar{A} + \bar{B} + \bar{C})(A + \bar{B})$
(*c*) $(A + C)(A\bar{B} + AC)(\bar{A}\bar{C} + \bar{B})$

3.16 Convert the following expressions to sum-of-products form:
(*a*) $(\bar{A} + \bar{B})(\bar{C} + B)$ (*b*) $\bar{A}B(\bar{B}C + \bar{B}\bar{C})$
(*c*) $(A + B\bar{C})(\bar{A}\bar{B} + \bar{A}B)$ (*d*) $AB(A\bar{B}\bar{C} + \bar{A}C)$
(*e*) $(\bar{A} + B)[A\bar{C} + (B + C)]$ (*f*) $(\bar{A} + C)(AB + \bar{A}\bar{B} + AC)$

QUESTIONS

3.17 Which rule is the dual of rule 12 in Table 3.10?

3.18 Give a dual of the rule of $X + \overline{X}Y = X + Y$.

3.19 Multiply the following sum terms, forming a sum-of-products expression in each case. Simplify while multiplying when possible.

(*a*) $(A + C)(B + D)$
(*b*) $(A + C + D)(B + D + C)$
(*c*) $(AB + C + DC)(AB + BC + D)$
(*d*) $(A\overline{B} + \overline{A}B + A\overline{C})(\overline{A}\overline{B} + A\overline{B} + A\overline{C})$

3.20 Convert the following expressions to product-of-sums form:

(*a*) $A + \overline{A}B + \overline{A}\overline{C}$
(*b*) $BC + \overline{A}B$
(*c*) $A\overline{B}(\overline{B} + \overline{C})$
(*d*) $\overline{A}\overline{B}(\overline{B}\overline{C} + \overline{B}\overline{C})$
(*e*) $(A + \overline{B} + C)(AB + \overline{A}C)$
(*f*) $(\overline{A} + \overline{B})A\overline{B}C$

3.21 Write the boolean expression (in sum-of-products form) for a logic circuit that will have a 1 output when $X = 0, Y = 0, Z = 1$ and $X = 1, Y = 1, Z = 0$; and a 0 output for all other input states. Draw the block diagram for this circuit, assuming that the inputs are from flip-flops.

3.22 Convert the following to product-of-sums form

(*a*) $AB + \overline{A}(B + \overline{C})(D + \overline{B})$
(*b*) $(B + C)[(\overline{B} + \overline{C})(A + \overline{C})(B + C)]$

3.23 Convert the following to product-of-sums form

(*a*) $AB\overline{C} + A\overline{B}C + \overline{A}BC$
(*b*) $ABC + \overline{B}\overline{C}(A + CD)(B + C)$

3.24 Prove the following theorem, using the rules in Table 3.10:

$$(X + Y)(X + \overline{Y}) = X$$

3.25 Write the boolean expression (in sum-of-products form) for a logic network that will have a 1 output when $X = 1, Y = 0, Z = 0$; $X = 1, Y = 1, Z = 0$; $X = 1, Y = 1, Z = 0$; and $X = 1, Y = 1, Z = 1$. The circuit will have a 0 output for all other sets of input values. Simplify the expression derived and draw a block diagram for the simplified expression.

3.26 Derive the boolean algebra expression for a gating network that will have outputs 0 only when $X = 1, Y = 1, Z = 1$; $X = 0, Y = 0, Z = 0$; $X = 1, Y = 0, Z = 0$. The outputs are to be 1 for all other cases.

3.27 Prove rule 18 in Table 3.10, using the proof by perfect induction.

3.28 Develop sum-of-products and product-of-sums expressions for F_1, F_2, and F_3 in Table 3.27.

3.29 Develop both the sum-of-products and the product-of-sums expressions that describe Table 3.28. Then simplify both expressions. Draw a block diagram for logical circuitry that corresponds to the simplified expressions, using only NAND gates for the sum-of-products and NOR gates for product-of-sums expression.

TABLE 3.27

INPUTS			OUTPUTS		
X	Y	Z	F_1	F_2	F_3
0	0	0	0	0	1
0	0	1	0	1	1
0	1	0	1	1	1
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	0	1	0
1	1	0	1	1	1
1	1	1	1	0	1

TABLE 3.28

INPUTS			OUTPUT
X	Y	Z	A
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

3.30 Draw block diagrams for the F_1, F_2, and F_3 in Question 3.28, using only NAND gates.

3.31 Write the boolean algebra expressions for Tables 3.29 to 3.31, showing expressions in sum-of-products form. Then simplify the expressions and draw a block diagram of the circuit corresponding to each expression.

3.32 Draw block diagrams for the F_1, F_2, and F_3 in Question 3.28, using only NOR gates.

3.33 Draw block diagrams for F_1, F_2, and F_3 in Question 3.28, using OR-to-NAND networks.

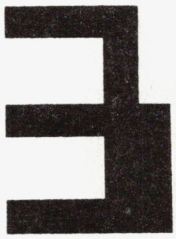
3.34 Draw block diagrams for F_1, F_2, and F_3 in Question 3.28, using AND-to-NOR gate networks.

3.35 Draw Karnaugh maps for the expressions in Question 3.2.

3.36 Draw Karnaugh maps for the expressions in Question 3.3.

3.37 For a four-variable map in W, X, Y, and Z draw the subcubes for:
(*a*) $WX\overline{Y}$ (*b*) WX (*c*) $XY\overline{Z}$ (*d*) Y

3.38 For a four-variable map in W, X, Y, and Z draw the subcubes for:
(*a*) $\overline{W}X\overline{Y}\overline{Z}$ (*b*) $W\overline{Z}$ (*c*) $\overline{W}Z$ (*d*) $\overline{Y}$



QUESTIONS

TABLE 3.29

INPUTS			OUTPUT
A	*B*	*C*	*Z*
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

3.39 Draw maps of the expressions in Question 3.40; then draw the subcubes for the shortened terms you found.

3.40 Apply the rule $AY + A\bar{Y} = A$, where possible, to the following expressions:

(*a*) $X\bar{Y} + \bar{X}\bar{Y}$ (*b*) $\bar{A}\bar{B}\bar{C} + A\bar{B}\bar{C}$

(*c*) $A\bar{B}C + ABC$ (*d*) $ABC + A\bar{B}\bar{C} + A\bar{B}C + AB\bar{C}$

(*e*) $ABC + \bar{A}\bar{B}\bar{C} + A\bar{B}C$ (*f*) $ABC + \bar{A}BC + \bar{A}B\bar{C}$

Note: There is a technique for writing minterms that is widely used. It consists in writing the letter *m* (to represent *minterm*) along with the value of the binary number given by the row of the table and combinations in which the minterm lies. For instance, in the variables *X*, *Y*, *Z* we have the unfinished table of combinations given in Table 3.32.

TABLE 3.30

INPUTS			OUTPUT
A	*B*	*C*	*Z*
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

TABLE 3.31

INPUTS			OUTPUT
X	*Y*	*Z*	*P*
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

TABLE 3.32

INPUT			OUTPUT	PRODUCT TERMS	DESIGNATION AS m_i
X	*Y*	*Z*			
0	0	0		$\bar{X}\bar{Y}\bar{Z}$	m_0
0	0	1		$\bar{X}\bar{Y}Z$	m_1
0	1	0		$\bar{X}Y\bar{Z}$	m_2
0	1	1		$\bar{X}YZ$	m_3
1	0	0		$X\bar{Y}\bar{Z}$	m_4
1	0	1		$X\bar{Y}Z$	m_5
1	1	0		$XY\bar{Z}$	m_6
1	1	1		XYZ	m_7

For this table $m_0 = \overline{X}\overline{Y}\overline{Z}$, $m_1 = \overline{X}\overline{Y}Z$, $m_2 = \overline{X}Y\overline{Z}$, $m_3 = \overline{X}YZ$, and so to $m_7 = XYZ$. Now we can substitute m_i's for actual terms and shorten the writing of expressions. For instance, $m_1 + m_2 + m_4$ means $\overline{X}\overline{Y}Z + \overline{X}Y\overline{Z} + X\overline{Y}\overline{Z}$. Similarly, $m_0 + m_3 + m_5 + m_7$ means $\overline{X}\overline{Y}\overline{Z} + \overline{X}YZ + X\overline{Y}Z + XYZ$.

This can be extended to four or more variables. An expression in W, X, Y, Z can be written as $m_0 + m_{13} + m_{15} = \overline{W}\overline{X}\overline{Y}\overline{Z} + WX\overline{Y}Z + WXYZ$. Or $m_2 + m_5 + m_9 = \overline{W}\overline{X}Y\overline{Z} + \overline{W}X\overline{Y}Z + W\overline{X}\overline{Y}Z$. As can be seen, to change a minterm to its m_i, simply make uncomplemented variables 1s and complemented variables 0s. Thus $\overline{W}XY\overline{Z}$ would be 0110, or 6 decimal; $\overline{W}\overline{X}Y\overline{Z}$ would be 0010, or 2 decimal. These two terms would then be written m_6 and m_2. (Notice that we must know how many variables a minterm is in.)

3.41 Draw the Karnaugh maps in X, Y, Z for:

(*a*) $m_0 + m_1 + m_5 + m_7$ (*b*) $m_1 + m_3 + m_5 + m_4$
(*c*) $m_1 + m_2 + m_3 + m_5$ (*d*) $m_0 + m_5 + m_7$

3.42 Draw the subcubes for a three-variable map in X, Y, Z for:

(*a*) $m_1 + m_3 + m_5 + m_0$ (*b*) $m_4 + m_7$ (*c*) $m_0 + m_3$

3.43 Find the maximal subcubes for the maps drawn for Question 3.42.

3.44 Find minimal expressions for the maps drawn in Question 3.42.

3.45 Using maps, simplify the following expressions in four variables, W, X, Y, and Z:

(*a*) $m_2 + m_3 + m_5 + m_6 + m_7 + m_9 + m_{11} + m_{13}$
(*b*) $m_0 + m_2 + m_4 + m_8 + m_9 + m_{10} + m_{11} + m_{12} + m_{13}$

3.46 Using maps, simplify the following expressions in four variables W, X, Y, and Z:

(*a*) $m_1 + m_3 + m_5 + m_7 + m_{12} + m_{13} + m_8 + m_9$
(*b*) $m_0 + m_5 + m_7 + m_8 + m_{11} + m_{13} + m_{15}$

3.47 Using maps, derive minimal product-of-sums expressions for the functions given in Question 3.46.

3.48 Using maps, derive minimal product-of-sums expressions for the functions given in Question 3.42.

3.49 Using maps, simplify the following expressions, using sum-of-products form:

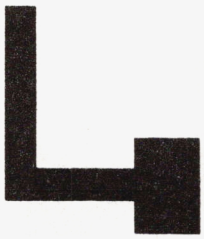
(*a*) $\overline{A}\overline{B}\overline{C} + A\overline{B}\overline{C} + \overbrace{ABC + \overline{A}B\overline{C} + \overline{A}\overline{B}C}^{\text{don't-cares}}$

(*b*) $ABC + \overline{A}\overline{B}\overline{C} + \overbrace{AB\overline{C} + A\overline{B}C}^{\text{don't-cares}}$

(*c*) $ABCD + \overline{A}\overline{B}\overline{C}D + \overline{A}BCD + \overbrace{A\overline{B}CD + \overline{A}\overline{B}CD + ABC\overline{D}}^{\text{don't-cares}}$

3.50 Using maps, derive minimal product-of-sums expressions for the functions given in Question 3.49.

3.51 Using maps, simplify the following expressions, using sum-of-products form:



QUESTIONS

4.1 Draw a set of waveforms for S and R and X and $\overline{X}$ (as in Fig. 4.2) so that the flip-flop in Fig. 4.1 will have the output signals 0011010 on the output line.

4.2 If the AND gate connected to the R input of X_1 in Fig. 4.3 fails so that its output is always 1, we would expect, after a few transfers, that X_1 would always be in what state? Why?

4.3 Draw a set of waveforms for S and R (as in Fig. 4.2) so that the flip-flop in Fig. 4.1 will have the output signals 101110001 on the X output line.

4.4 If the $\overline{X}$ output of flip-flop X is connected to an inverter, the inverter's output will always be the same as the X output of the flip-flop. True or false? Why?

4.5 Draw a set of waveforms as in Fig. 4.5(*d*) for the flip-flop in Fig. 4.5(*b*) so that the flip-flop will have the output signals 0010110 on its Y output line.

4.6 In Fig. 4.10 if flip-flop X_2 "sticks" (that is, fails) in its 0 state, will X_3 have a 1 output after (*a*) clock pulse 1, (*b*) clock pulse 2, (*c*) clock pulse 3 and for each clock pulse thereafter?

4.7 Draw an input waveform as in Fig. 4.10 so that X_3 will have the output signal 010011010 if X_1, X_2, and X_3 are started in the 0 state.

4.8 Draw a set of waveforms as in Fig. 4.5(*c*) for the flip-flop in Fig. 4.5(*a*) so that the flip-flop will have the output signals 10111001 on its X output line.

4.9 The binary counter in Fig. 4.11 uses flip-flops which act on positive transitions. Draw a block diagram of a binary counter with bubbles on the clock inputs (that is, with flip-flops which act on *negative*-going clock inputs).

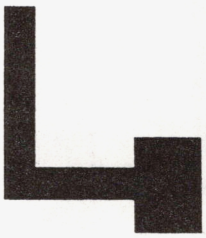
4.10 If the X_3 line (the output of X_3) is connected to the input line in Fig. 4.10, a *ring counter* is formed. If this circuit is started with $X_1 = 0$, $X_2 = 1$, and $X_3 = 1$, draw the waveform at X_1, X_2, and X_3 for six clock pulses.

4.11 Redraw Fig. 4.11 as it is, but place bubbles on each clock input to X_1, X_2, and X_3 [that is, make the same drawing, but use the flip-flop in Fig. 4.5(*b*) instead of that in Fig. 4.5(*a*)]. Redraw the waveforms in Fig. 4.11 for this circuit.

4.12 Does the counter in Question 4.11 count up or down?

4.13 Make a single change in Fig. 4.11 by connecting the output of flip-flop X_2 to the clock input of X_3 instead of the $\overline{X}_2$ output of X_2. Now redraw the waveforms in Fig. 4.11 for this changed configuration.

4.14 After answering Question 4.13, using the flip-flop in Fig. 4.5(*a*), design a counter that counts as follows:



QUESTIONS

X_3	X_2	X_1
0	0	0
0	1	1
0	1	0
1	0	1
1	0	0
1	1	1
1	1	0
0	0	1
0	0	0
0	1	1
0	1	0
1	0	1
.	.	.

4.15 After answering Question 4.13, using only the flip-flop in Fig. 4.5(*a*), design a counter that counts as follows:

X_3	X_2	X_1
0	0	0
1	1	1
1	1	0
1	0	1
1	0	0
0	1	1
0	1	0
0	0	1
0	0	0
1	1	1

This counter counts *down*. Figure 4.11 counts *up*.

4.16 For Fig. 4.14(*b*) draw UP ENABLE, DOWN ENABLE, and clock waveforms so that the counter starts at 000 and counts as follows:

X_3	X_2	X_1
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
0	1	1
0	1	0
0	1	1

4.17 In Fig. 4.14(*a*), when the ENABLE line is a 1, the counter counts at the occurrence of a negative-going clock edge. Draw the output waveforms for FF0, FF1, and FF2 for these waveforms. Start the flip-flops at 0.

188

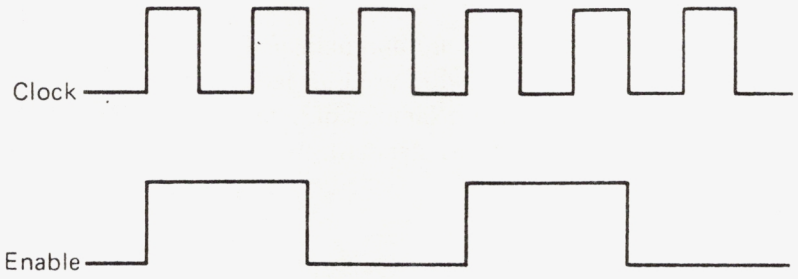


FIGURE Q4.17

4.18 Draw a clock waveform and waveforms for the output of the AND gate connected to the S input of X_1, the R input of X_2, and the 1 output of X_1, X_2, and X_3 for Fig. 4.12.

4.19 Suppose that the AND gate in Fig. 4.15(a) fails, so that its output is always a 0. Write the sequence or show the waveform through which this counter will go in response to clock signals.

4.20 Suppose that the AND gate's output in Fig. 4.15(a) is connected to the K input of X_4 instead of the J input. How will the computer count?

4.21 For Fig. 4.15(b), the carry-out from block 1 to block 2 goes from 1 to 0 every _____ clock pulses. The carry-out from block 3 to block 4 goes from 1 to 0 every _____ clock pulses.

4.22 (a) If we replace block 2 in Fig. 4.15(b) with the four-stage ripple counter in Fig. 4.14(a), using the 1 output of FF3 in that counter as the carry-out line, the carry-out line from block 2 will go from 1 to 0 after how many clock pulses?

(b) For the configuration in (a), after how many clock pulses will the carry-out from block 2 go from 0 to 1?

4.23 Using the circuits in Fig. 4.16, design a gated-clocked binary counter.

4.24 Design a BCD counter using the flip-flops in Fig. 4.16.

4.25 Using the circuits in Fig. 4.16, design a gating network with inputs A, B, and C which will have output 1 when $\bar{A}\bar{B}C$ or $A\bar{B}\bar{C}$ are 1s.

4.26 Using the circuits in Fig. 4.16, design a BCD counter.

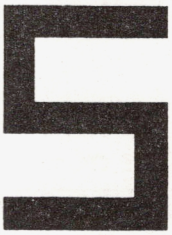
4.27 Using the circuits in Fig. 4.16, design a gate network with inputs A, B, and C and output $A\bar{B} + \bar{A}C$.

4.28 Does a four-input NAND gate, as in Fig. 4.16, with the third input held at 1, act as a three-input gate would if only two inputs were used? Explain your answer.

4.29 Design the counter in Fig. 4.12, using the blocks in Fig. 4.16.

4.30 Give the states of the flip-flops in the following circuit after each of the first five clock signals (pulses) are applied. The circuit is started in the state $A_1 = 0, A_2 = 0, A_3 = 1$.

Often calculations are not exact since numbers are approximated in many cases, but floating-point number systems are almost always used for large scientific computations and by higher-level languages. Several examples of floating-point number systems were explained, including the new ANSI standard.



QUESTIONS

QUESTIONS

5.1 Draw a block diagram of circuitry for two registers X and Y of three flip-flops each, so that Y can be transferred into X, or the 1s complement of Y can be transferred into X.

5.2 Two parallel binary registers, designated register X and register Y, both consist of three flip-flops. Draw a block diagram of the registers and the necessary logic circuitry so that (*a*) register X can be cleared, or 1s complemented, and (*b*) the 1s complement of the contents of register Y can be transferred into register X.

5.3 If a binary computer handles numbers in the sign-plus-magnitude integer system and numbers are 5 bits (sign plus 4 bits) each, how would the following decimal numbers be represented? For example, $+5 = \underline{0}0101$.

(*a*) $+6$ (*b*) $+10$ (*c*) -12 (*d*) -16

5.4 If a binary computer represents numbers in a sign-plus-magnitude form with 5 bits per number, how would the following decimal numbers be represented.

(*a*) $+8$ (*b*) $+11$ (*c*) -7
(*d*) -4 (*e*) -15 (*f*) -12

5.5 If a binary machine handles negative numbers in the true magnitude form, how would -4 be stored in a register with a sign bit and 4 bits representing magnitude? If the same machine stored numbers in the 1s complement system, how would -4 be stored?

5.6 If a register contains five flip-flops as in Fig. 5.1(*b*) and the register contains $X_0 = 1, X_2 = 1, X_3 = 0, X_4 = 0, X_5 = 1$, give the decimal value of the number in the register if the 1s complement number system is used. What is the value if 2s complement is used?

5.7 The inputs to the full adder in Fig. 5.3 are as follows: $X = 1$, $Y = 1$, and $C = 1$. What will the output on the S and C lines represent?

5.8 If we use the $\oplus$ symbol to mean exclusive OR and define it as $X \oplus Y = \bar{X}Y + X\bar{Y}$, then the output S from a full-adder can be written as $S = X \oplus Y \oplus C_i$. Show why this is the case.

5.9 If we load the binary number $\underline{1}0011$ into the flip-flops in Fig. 5.1(*b*), that is, if $X_4 = 1, X_3 = 0, X_2 = 0, X_1 = 1$, and $X_0 = 1$, what will the value of the register be in the 1s complement number system? Give the answer in decimal. What will the value of this number be if the 2s complement number system is used?

5.10 If register X contains $\underline{0}0111$ and register Y contains $\underline{1}1011$, what do the two numbers represent in decimal if 1s complement is used? If 2s complement is used?

5

THE ARITHMETIC-LOGIC UNIT

Add the two numbers in 1s complement, then in 2s complement, and give the results in decimal.

5.11 Register X contains $\underline{0}1100$, and register Y contains $\underline{0}1101$ (where the underscored 0s designate that the number stored is positive). If the two registers are added, what will the result be?

5.12 If 1s complement is used, for which of the following expressions will an overflow or end-around carry be generated? Why? Assume 5-bit registers, including a sign bit.

$(a)\ +5+(-7)$ $\quad$ $(b)\ +5+(-4)$

$(c)\ +12+(-13)$ $\quad$ $(d)\ +12+3$

5.13 Add a stage to the load-and-shift register in Fig. 5.17. Copy only D in your drawing, omitting A, B, and C.

5.14 A binary register consists of five binary storage devices; one stores the sign bit, and the other four store the magnitude bits. If the number stored is $\underline{0}0110$ and this number is then shifted right one binary place, what will be the result? Assume a 0 goes into the sign bit.

5.15 Design a half-adder, using only NOR gates.

5.16 Design a half-adder, using only NAND gates.

5.17 A binary half-subtracter has two inputs x and y and two outputs, which are the "difference" value $x - y$ and a "borrow" output that is 1 if the value of $x - y$ is negative ($x - y$ is then given the value 1). Draw a block diagram for a half-subtracter, using NAND gates and assuming x, $\bar{x}$, y, and $\bar{y}$ are all available as inputs.

5.18 Design a half-subtracter, using only NOR gates.

5.19 If a borrow input is added to the half-subtracter in Questions 5.17 and 5.18, a full-subtracter is formed. Design a full-subtracter, using only NAND gates.

5.20 Design a full-subtracter, using only NOR gates.

5.21 Design a full-adder, using only NAND gates.

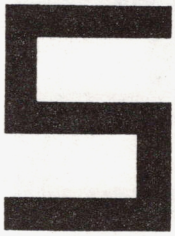
5.22 Can an overflow occur during multiplication in a binary machine with numbers stored in fixed-point sign-plus-magnitude form? Assume a double-length product.

5.23 Explain how the 2s complement of the subtrahend is formed when we subtract, using the configuration in Fig. 5.6. How is the 1 added in to form this complement?

5.24 Show how the configuration in Fig. 5.6 adds and subtracts by adding and subtracting $+21$ and $+3$ in binary, showing what each output will be.

5.25 Using the configuration in Fig. 5.6, add and subtract $+6$ and -4, showing what each output will be and checking the correctness of the sum and difference.

5.26 Draw a multiplexer using only NAND gates which selects from four inputs I_0 to I_3, using two select inputs S_0 and S_1.



QUESTIONS

5.27 Design a multiplexer for four inputs, using a two-level NOR gate combinational network. The inputs are to be X_0, X_1, X_2, and X_3. The output is to be called W. The X_0, X_1, X_2, and X_3 are "selected" by S_0 and S_1. If S_0 and $S_1 = 0$, then W should equal X_0. If S_1 and $S_0 = 01$, then W should equal X_1. If S_0 and $S_1 = 10$, then W should equal X_2. If S_0 and $S_1 = 11$, then W should equal X_3.

5.28 Add a control line NAND and circuitry to Fig. 5.20 so that the NAND of ACC and B can be transferred into AOC.

5.29 If a register containing 0110011 is logically added to a register containing 0101010, what will the result be? What will be the result if the registers are logically multiplied? If the registers are exclusive ORed?

5.30 Find a sequence of logical operations which will cause the ACC to have value 0 regardless of how ACC and B start in Fig. 5.20.

5.31 Referring to Fig. 5.20, show that a logical multiplication followed by a logical addition will transfer the contents of B to ACC.

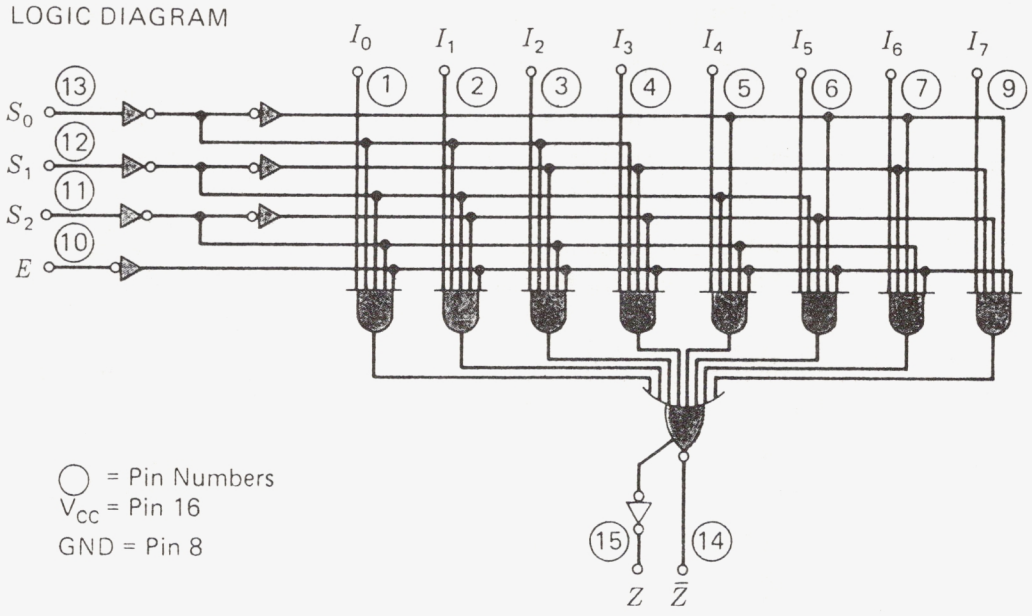
5.32 Add a control line NOR and circuitry to Fig. 5.20 so that the NOR of ACC and B can be transferred into ACC.

5.33 Demonstrate by means of a table of combinations that two half-adders plus an OR gate do make a full-adder, as shown in Fig. 5.4.

5.34 Add gates and a flip-flop X_{m+2} so that we can shift left and right into X_{m+1} in Fig. 5.16.

5.35 Add gates to the following drawing so that if the control input signal C is made a 1, the value $\overline{X}\overline{Y}$ will appear on the output line. If A is a 1, then $X + Y$ appears on the output; if B is a 1, then $X \cdot Y$ appears on the output; if A, B, and C are 0s, then the output is to be a 0; and only one of A, B, or C can be a 1 at a given time.

FIGURE 5.35



5

THE ARITHMETIC-LOGIC UNIT

5.36 In what case will subtracting a negative number from 0 cause an overflow in the 2s complement system?

5.37 Design a *parallel multiplier* gate network with two inputs A_1A_0 and $B_2B_1B_0$, where A_1A_0 are two binary digits forming a binary number (with decimal values for 0 to 3) and $B_2B_1B_0$ is a 3-bit binary number (with values 0 to 7). Use only AND gates and OR gates, and see that no signal passes through more than four levels of gates. (Assume input complements are available.)

5.38 Write the boolean algebra expressions for the four least significant bits in a parallel multiplier which multiplies two 4-bit binary numbers.

5.39 Design a gating network for a module in an ALU that will add two 2-bit binary inputs A_j, A_{j-1} and B_j, B_{j-1} and an input carry-in C_{j-1}. The network is to generate the two sum digits and a carry-out C_{j+2}. Use only AND or OR gates and inverters, but assume that both complemented and uncomplemented inputs are available. The output carry C_{j+2} should have a delay of no more than three-gate delays (that is, a change in an input must pass through no more than three gates in any path to an output).

5.40 (*a*) Give the IBM floating-point representation for decimal 27.25 and -27.25.
(*b*) Give the IEEE standard format representation for $+12$ and -12.

5.41 (*a*) Give the IBM floating-point representation for $+55.4$ and -53.4.
(*b*) Give the IEEE standard single-precision floating-point representation for $+29$ and -29.
(*c*) Compare the ranges, accuracy, and other system design considerations for the above two computer floating-point number systems.

5.42 If numbers are represented in a 2s complement, 7-bit magnitude plus 1 sign-bit integer system, and we ignore overflow (that is, any result will be stored, even if it requires more magnitude bits), then the largest positive integer which can result from the addition of two numbers is _____, and the largest positive integer which can result from the subtraction of one number from another is _____. The least negative numbers which can result from an addition and subtraction are _____ and _____. As a result, any sum or difference can be stored in _____ bits.

5.43 Write all 4-bit 2s complement numbers (that is, sign plus 3-bit numbers) and their decimal values. Show that there is one more negative number than positive number. Consider 0 to be neither negative nor positive.

5.44 When addition is performed in a binary machine using the 2s complement number system to represent negative numbers, an overflow may occur in a register only when two positive or two negative numbers are added. Show that the addition of a positive number and a negative number cannot result in an overflow condition.

5.45 Show that there are as many negative as positive numbers in a 1s complement system.

5.46 The boolean algebra expressions on the output lines from the gates in Fig. 5.8 are not filled in. Develop the boolean algebra expressions for the *S*1, *S*2, and *C*2 outputs from the network.

5

QUESTIONS

5.47 Which of the following number systems has two 0s?
(*a*) Sign plus magnitude (*b*) 1s complement
(*c*) 2s complement

5.48 When the 2s complement number system is used and addition is performed, let us designate the carry-out of the full-adder connected into the full-adder for the sign digits as C_3 (refer to Fig. 5.6). The rule for overflow is that two numbers cause an overflow when they are added either if both numbers are positive and C_3 equals a 1 or if both numbers are negative and C_3 does not equal a 1. Therefore, by examining the sign digits of the two numbers being added and the carry-out of the full-adder which adds the two most significant digits of the magnitude of the numbers, we can form a logic network whose output will be a 1 when an overflow condition arises and a 0 if the addition is legitimate. Let X_4 store the sign digit of the addend, let Y_4 store the sign digit of the augend, and let C_3 again be the carry-out of the full-adders connected to the X_3 and Y_3 flip-flops, as in Fig. 5.6. Show that the logic equation for an overflow condition is $X_4Y_4\overline{C}_3 + \overline{X}_4\overline{Y}_4C_3 =$ overflow.

5.49 Modify Fig. 5.16 so that the complement of X can be shifted into X_m, that is, so that SHIFT LEFT causes $\overline{X}_{m-1}$ to go into X_m.

5.50 Show that when we add 7 to 9 in the BCD system using the series-parallel BCD adder in Fig. 5.10, the answer will be correct. Do this by tracing the outputs of the circuit, filling in the binary value for each X and Y shown in the figure, and by showing the values of Z_8, Z_4, Z_2 and Z_1.

5.51 Show how the BCD adder in Fig. 5.11 adds +6 to +5 by calculating each output from the gates and then the final outputs.

5.52 Check how the gates in Fig. 5.12 form a 9s complement by trying 5 and 3 in BCD at the inputs.

5.53 Explain the operation of Fig. 5.15 by explaining how 234 can be added to or subtracted from 523 in this configuration.

5.54 What is the function of the D flip-flop in Fig. 5.15?

5.55 Explain how to load the input values on W, X, Y, and Z into the flip-flops in Fig. 5.17.

5.56 Explain how to cause the flip-flops in Fig. 5.17 to shift right three times. Suppose we (1) make the *reset* input a 0 then a 1, (2) hold J and K at 1 and shift at 1, and (3) apply three clock pulses to the CLOCK line. Draw the output waveforms for A, B, C, and D for this sequence of inputs.

5.57 What is the result if we multiply 01101 times 00011 in our generalized machine in Fig. 5.18? Give the values in each X and B flip-flop.

5.58 What is the binary number that represents −3 in the 2s complement fractional number system if we represent the number by using a sign digit plus four magnitude digits?

5.59 Using the 8,4,2,1 BCD system with a single digit for the sign digit, write

memory are read into a secondary (disk or tape) memory. When the power is returned, the data are reentered into the IC memory. It is important that the power not decline too much before the transfer of information can be made. As a result, it is generally necessary to have some sort of backup power to carry the memory through until the contents of the IC memory can be read out. Calculate how long this would take for a 256K memory with a cycle time of 500 ns.

6.10 Question 6.9 concerned dumping an IC memory on a disk in case of power failure. It is also necessary to find a spare area on the disk and to transfer the contents of the memory onto the disk. For one of the disk memories described in the chapter, discuss this problem for the preceding 256K 500-ns IC memory.

6.11 Several microcomputers come with a basic 256K-word 8-bit memory. How many flip-flops are in (*a*) the memory address register and (*b*) the memory buffer register?

6.12 Draw the waveforms for recording the binary sequence 101, showing the signal applied to the write-head winding and the signal at the read-head winding for the type 1, type 2, and type 3 nonreturn-to-zero recording techniques.

6.13 A block diagram of an INTEL MOS LSI memory chip is shown in Fig. 6.10. Lines A_1 through A_8 are for the 256 addresses, and R/W tells whether to read or to write (a 0 on this line is READ, a 1 is WRITE). The CHIP SELECT disables the memory for a 1 input and enables the memory with a 0 output. Discuss the construction of a 256-word 8-bit memory using these packages. How many packages are required? Each flip-flop in the address register (external) would have its 1 output connected to how many chips?

6.14 Draw the schematic for a many-to-one decoder matrix with inputs from 4 flip-flops and 16 output lines. Use the same basic configuration as is illustrated in Fig. 6.5(*a*).

6.15 The INTEL package in Fig. 6.10 is representative of several IC manufacturers' products. Show how the CHIP SELECT can be used to add words to a memory.

6.16 What are the primary advantages of bipolar memory over MOS memory?

6.17 Explain the operation of the digit line in a linear-select core memory.

6.18 Explain the difference between a dynamic and a static MOS memory.

6.19 Contrast the parallel tree and balanced decoder networks for a 32-output decoder. Figure the number of diodes used for each and the delay incurred because of the number of gates a signal must pass through for each, assuming diode AND gates are used.

6.20 Given eight 8-bit registers A, B, C, . . ., H, show how a transfer circuit can be made by using multiplexers so that the contents of any register can be selected by a 3-bit register S and transferred to an 8-bit register X.

6.21 In larger machines, when the ac power drops below a certain level, the contents of the control unit and arithmetic-logic unit are dumped on tape or disks

so that the computer can be restarted with no loss of data. If IC memories are used, their contents must be dumped also. Explain why.

6.22 Explain the operation of the memory cell in Fig. 6.3.

6.23 Explain the operation of the memory cell in Fig. 6.8.

6.24 When linear selection is used for IC memories, individual cells tend to be simpler than for two-dimensional cell select systems, but the decoders tend to be more complicated. Explain why.

6.25 Why are floppy disks more used than hard disks in microcomputers?

6.26 The memory in Fig. 6.19 uses the organization in Fig. 6.11 and has linear-select memory cells. This simplifies individual memory cell complexity while increasing sense and write circuitry. Explain why this is cost-effective for a memory of this type.

6.27 Dynamic memories that require external refreshing introduce extra complexity into computer operation. Why?

6.28 Explain the advantages and disadvantages of dynamic IC memories.

6.29 In designing a 256-word 8-bit memory, pin 1 is connected to pin 1 for each container of the chip shown in Fig. 6.10. This applies to all *A* inputs and to chip-select bits and R/W, but not to data out or data in. Why?

6.30 Show how to expand the 256-word 1-bit memory in Fig. 6.10 to a 2048-word 1-bit memory by using the CHIP SELECT and a three-input, eight-output decoder.

6.31 Which of the tape units described will transfer data fastest, and how fast can characters and bits be read or written?

6.32 Make up a formula to calculate how many bits a second can be read from a disk memory that revolves at a rate of r revolutions per second and has b bits per track. What will the average latency time be?

6.33 Assuming that a disk with 1500 bits/in. rotates at a speed of 2400 rpm and that we read from eight tracks simultaneously, how many bits per second can be read from one of these disks?

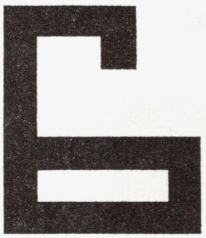
6.34 Explain seek time and latency for disk memories.

6.35 A magnetic-tape system has seven tracks for each $\frac{1}{2}$-in. width of tape. The packing density per track is 1280 bits/in., and the tape is moved at a speed of 75 in./s. If the tape width is 0.5 in., how many bits may be read per second?

6.36 Fixed-head disk memories reduce total access time by avoiding either seek time or latency. Which is avoided and why?

6.37 Contrast the floppy-disk figures for the 5- and the 8-in. systems.

6.38 Tape cassettes and tape cartridges have advantages and disadvantages compared to conventional tape systems. Cite several of each.



QUESTIONS

1.32

ADDRESS	OPERATION	OPERAND	ADDRESS	OPERATION	OPERAND
1	CLA	300	13	CLA	6
2	BRM	5	14	ADD	400
3	STO	300	15	STO	6
4	BRA	7	16	CLA	401
5	SUB	300	17	ADD	400
6	SUB	300	18	STO	401
7	CLA	3	19	BRM	1
8	ADD	400	20	HLT	
9	STO	3	300–329	contains numbers	
10	CLA	5			
11	ADD	400	400	contains 1	
12	STO	5	401	contains −30	

Chapter 2

2.1 (*a*) 101011 (*b*) 1000000 (*c*) 100000000000 (*d*) 0.011 (*e*) 0.11011 (*f*) 0.0111 (*g*) 1000000000.1 (*h*) 10000011.1001 (*i*) 10000000000.0001

2.3 (*a*) 13 (*b*) 27 (*c*) 23 (*d*) 0.6875 (*e*) 0.203125 (*f*) 0.212890625 (*g*) 59.6875 (*h*) 91.203125 (*i*) 22.3408203125

2.5 (*a*) 11 (*b*) 36 (*c*) 19 (*d*) 0.8125 (*e*) 0.5625 (*f*) 0.3125 (*g*) 11.1875 (*h*) 9.5625 (*i*) 5.375

2.7 (*a*) 10100.11 = 20.75 (*b*) 1001010 = 74 (*c*) 1.1 = 1.5 (*d*) 10101 = 21

2.9 (*a*)
$$\begin{array}{r} 1101.1 \\ \underline{1011.1} \\ 11001.0 \end{array} \qquad \begin{array}{r} 13.5 \\ \underline{11.5} \\ 25 \end{array}$$

(*b*)
$$\begin{array}{r} 101101 \\ \underline{110110} \\ 1100011 \end{array} \qquad \begin{array}{r} 45 \\ \underline{109} \\ 154 \end{array}$$

(*c*)
$$\begin{array}{r} 0.0011 \\ \underline{0.1110} \\ 1.0001 \end{array} \qquad \begin{array}{r} 0.1875 \\ \underline{0.875} \\ 1.0625 \end{array}$$

(*d*)
$$\begin{array}{r} 1100.011 \\ \underline{1011.011} \\ 10111.110 \end{array} \qquad \begin{array}{r} 12.375 \\ \underline{11.375} \\ 23.750 \end{array}$$

2.11 (*a*)
$$\begin{array}{r} 1000000 \\ \underline{-100000} \\ 100000 \end{array}$$

(*b*)
$$\begin{array}{r} 1111111 \\ \underline{-111111} \\ 1000000 \end{array}$$

(*c*)
$$\begin{array}{r} 1011101.1 \\ \underline{-101010.11} \\ 110010.11 \end{array}$$

(*d*)
$$\begin{array}{r} 1010100.01001 \\ \underline{-110000.01010} \\ 100011.11111 \end{array}$$

2.13 (*a*)
$$\begin{array}{r} 100101 \\ \underline{-100011} \\ 000010 \end{array}$$

(*b*)
$$\begin{array}{r} 10000000 \\ \underline{01000000} \\ 1000000 \end{array}$$

(*c*)
$$\begin{array}{r} 1011110.1 \\ \underline{101011.11} \\ 110010.11 \end{array}$$

(*d*)
$$\begin{array}{r} 11111111 \\ \underline{1111111} \\ 10000000 \end{array}$$

2.15 (*a*) 100100000 (*b*) 11111100 (*c*) 100 (*d*) 1.1 (*e*) 10100000001.101 (*f*) 10.1

2.17 (*a*)
$$\begin{array}{r} 1111 \\ \times\ 1101 \\ \hline 1111 \\ 11110 \\ 1111 \\ \hline 11000011 \end{array}$$

(*b*)
$$\begin{array}{r} 1111 \\ \times\ 1010 \\ \hline 11110 \\ 11110 \\ \hline 10010110 \end{array}$$

(*c*)
$$\begin{array}{r} 100 \\ 1011\overline{)101100} \\ 1011 \\ \hline 000 \end{array}$$

(*d*)
$$\begin{array}{r} 11.1 \\ 1100\overline{)101010.0} \\ 1100 \\ \hline 10010 \\ 1100 \\ \hline 1100 \\ 1100 \\ \hline 0 \end{array}$$

(*e*)
$$\begin{array}{r} 111.11 \\ \times\ 10.1 \\ \hline 11111 \\ 111110 \\ \hline 10011.011 \end{array}$$

(*f*)
$$\begin{array}{r} 10110.1 \\ \times\ 100.11 \\ \hline 101101 \\ 101101 \\ 10110100 \\ \hline 1101010.111 \end{array}$$

2.19

	9s COMPLEMENT	10s COMPLEMENT
(*a*)	4563	4564
(*b*)	8067	8068
(*c*)	54.84	54.85
(*d*)	81.706	81.707

2.21

	9s COMPLEMENT	10s COMPLEMENT
(*a*)	6345	6346
(*b*)	7877	7878
(*c*)	45.80	45.81
(*d*)	62.736	62.737

2.23

	1s COMPLEMENT	2s COMPLEMENT
(*a*)	0100	0101
(*b*)	00100	00101
(*c*)	0100.10	0100.11
(*d*)	00100.10	00100.11

2.25

	1s COMPLEMENT	2s COMPLEMENT
(*a*)	010000	010001
(*b*)	011011	011100
(*c*)	01000.01	01000.10
(*d*)	01100.00	

2.27

	9s COMPLEMENT	10s COMPLEMENT
(*a*)	948	948
	$\underline{765}$	$\underline{766}$
	1 713	714
	$\hookrightarrow \underline{1}$	
	714	

(*b*)

	347	347
	736	737
1	083	084
	→ 1	
	084	

(*c*)

	349.5	349.5
	754.6	754.7
1	104.1	104.2
	→ 1	
	104.2	

(*d*)

	412.7	412.7
	590.7	590.8
1	003.4	3.5
	→ 1	
	3.5	

2.29

(*a*)

	9s COMPLEMENT	10s COMPLEMENT
	1024	1024
	9086	9087
	0110	0111
	→ 1	
	111	

(*b*)

	249	249
	862	863
	111	112
	→ 1	
	112	

(*c*)

	24.1	24.1
	86.5	86.6
	10.6	10.7
	→ 1	
	10.7	

(*d*)

	239.3	239.3
	880.5	880.6
	119.8	119.9
	→ 1	
	119.9	

2.31

(*a*)

	1s COMPLEMENT	2s COMPLEMENT
	1011	1011
	1010	1011
1	0101	0110
	→ 1	
	0110	

(*b*)

	11011	11011
	00110	00111
1	00001	00010
	→ 1	
	10	

(*c*)
$$\begin{array}{r} 10111.1 \\ \underline{01100.0} \\ 1\ 00011.1 \\ \underline{\longrightarrow\ 1} \\ 100.0 \end{array} \qquad \begin{array}{r} 10111.1 \\ \underline{01100.1} \\ 100.0 \end{array}$$

(*d*)
$$\begin{array}{r} 11011.00 \\ \underline{01100.00} \\ 1\ 00111.00 \\ \underline{\longrightarrow\ 1} \\ 111.01 \end{array} \qquad \begin{array}{r} 11011.00 \\ \underline{01100.01} \\ 111.01 \end{array}$$

2.33 $2^6 = 64$

2.35 1000 different numbers in each case (from 0 to 999, for instance)

2.37 0, 1, 2, 3, 10, 11, 12, 13, 20, 21

2.39 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 1A, 20, 21, 22

2.41 (*a*)
$$\begin{array}{r} .1001 \\ \underline{.1001} \\ 0010 \\ \underline{\longrightarrow\ 1} \\ .0011 \end{array}$$
(*b*)
$$\begin{array}{r} .1110 \\ \underline{.1001} \\ 0111 \\ \underline{\longrightarrow\ 1} \\ .1000 \end{array}$$
(*c*)
$$\begin{array}{r} 01111 \\ \underline{10110} \\ 00101 \\ \underline{\longrightarrow\ 1} \\ .00110 \end{array}$$
(*d*)
$$\begin{array}{r} 11011 \\ \underline{00110} \\ 00001 \\ \underline{\longrightarrow\ 1} \\ 00010 \end{array}$$
(*e*)
$$\begin{array}{r} 1110101 \\ \underline{0101101} \\ 1\ 0100010 \\ \underline{\longrightarrow\ 1} \\ 0100011 \end{array}$$

2.45 (*a*)
$$\begin{array}{r} 45{,}056 \\ 1{,}536 \\ 192 \\ \underline{7} \\ 46{,}791 \end{array}$$
(*b*)
$$\begin{array}{r} 24{,}576 \\ 1{,}024 \\ 160 \\ \underline{12} \\ 25{,}772 \end{array}$$
(*c*)
$$\begin{array}{r} 40{,}960 \\ 1{,}024 \\ 144 \\ \underline{2} \\ 42{,}130 \end{array}$$
(*d*)
$$\begin{array}{r} 851{,}968 \\ 8{,}192 \\ 1{,}792 \\ 96 \\ \underline{3} \\ 862{,}051 \end{array}$$

2.49 (*a*) 55 (*b*) 556 (*c*) 267 (*d*) 66.3 (*e*) 3.554

2.53 (*a*) 1644. (*b*) 514 (*c*) 1041.3 (*d*) 1170.76051 (*e*) 10515.5

2.57 (*a*) B7 (*b*) 9C (*c*) 5F (*d*) 0.7E (*e*) B7A

2.61 (*a*)
$$\begin{array}{r} {\scriptstyle 1} \\ 15_8 \\ +\ 14_8 \\ \underline{} \\ 31_8 \end{array}$$
(*b*)
$$\begin{array}{r} {\scriptstyle 1} \\ 24_8 \\ +\ 36_8 \\ \underline{} \\ 62_8 \end{array}$$
(*c*)
$$\begin{array}{r} {\scriptstyle 1} \\ 126_8 \\ 347_8 \\ \underline{} \\ 475_8 \end{array}$$
(*d*)
$$\begin{array}{r} {\scriptstyle 1} \\ 67_8 \\ 45_8 \\ \underline{} \\ 134_8 \end{array}$$
(*e*)
$$\begin{array}{r} {\scriptstyle 1} \\ 136_8 \\ 636_8 \\ \underline{} \\ 774_8 \end{array}$$

588 **Chapter 3**

ANSWERS TO SELECTED ODD-NUMBERED QUESTIONS

3.1 (*a*)

X	Y	Z	XYZ	$X\overline{Y}\overline{Z}$	$XYZ + X\overline{Y}\overline{Z}$
0	0	0	0	1	1
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	0	0	0
1	0	1	0	0	0
1	1	0	0	0	0
1	1	1	1	0	1

(*b*)

A	B	C	ABC	$A\overline{B}\overline{C}$	$\overline{A}\overline{B}\overline{C}$	$ABC + A\overline{B}\overline{C} + \overline{A}\overline{B}\overline{C}$
0	0	0	0	0	1	1
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	0	0	0	0
1	0	0	0	1	0	1
1	0	1	0	0	0	0
1	1	0	0	0	0	0
1	1	1	1	0	0	1

(*c*)

A	B	C	$B\overline{C}$	$\overline{B}C$	$B\overline{C} + \overline{B}C$	$A(B\overline{C} + \overline{B}C)$
0	0	0	0	0	0	0
0	0	1	0	1	1	0
0	1	0	1	0	1	0
0	1	1	0	0	0	0
1	0	0	0	0	0	0
1	0	1	0	1	1	1
1	1	0	1	0	1	1
1	1	1	0	0	0	0

(*d*)

A	B	C	$A + B$	$A + C$	$\overline{A} + \overline{B}$	$(A + B)(A + C)(\overline{A} + \overline{B})$
0	0	0	0	0	1	0
0	0	1	0	1	1	0
0	1	0	1	0	1	0
0	1	1	1	1	1	1
1	0	0	1	1	1	1
1	0	1	1	1	1	1
1	1	0	1	1	0	0
1	1	1	1	1	0	0

3.3 Only the values of the expressions are listed:

(*a*)

A	B	$A\overline{B} + \overline{A}B$
0	0	0
0	1	1
1	0	1
1	1	0

(*b*)

A	B	C	$A\overline{B} + B\overline{C}$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

(*c*)

A	C	$A\overline{C} + AC$
0	0	0
0	1	0
1	0	1
1	1	1

(*d*)

A	B	C	$A\overline{B}C + AB\overline{C} + \overline{A}BC$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

(*e*)

A	B	C	$A(A\overline{B}C + A\overline{B}\overline{C} + AB\overline{C})$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

3.7 (*a*) $\overline{B}\overline{C} + \overline{A}\overline{C} + \overline{A}\overline{B}$ (*b*) $\overline{C} + \overline{B} + \overline{A}$
(*c*) $A(B + C)$ (*d*) A is a minimal expression

3.9 (*a*) $ABC(A\overline{B}\overline{C} + A\overline{B}C + \overline{A}BC) = 0$, for no assignment of binary values will make this expression take the value 0.

(*b*) $AB + A\overline{B} + \overline{A}C + \overline{A}\overline{C} = 1$, for every assignment of values will give this expression the value 1.

(*c*) $XY + XYZ + XY\overline{Z} + \overline{X}YZ = XY + YZ$

(*d*) $XY(\overline{X}YZ + X\overline{Y}\overline{Z} + \overline{X}\overline{Y}\overline{Z}) = 0$

3.11 (*a*) $\overline{A}(\overline{B} + \overline{C})(\overline{A} + \overline{B}) = \overline{A}\overline{B} + \overline{A}\overline{C}$ or $\overline{A}(\overline{B} + \overline{C})$

(*b*) $\overline{A}\overline{B} + \overline{B}\overline{C} + \overline{A}\overline{C}$

(c) $(\overline{A} + \overline{B})(B + \overline{C})(\overline{C} + D) = \overline{A}B\overline{C} + \overline{A}\overline{C} + \overline{A}BD + \overline{A}\overline{C}D$
$= \overline{A}\overline{C} + \overline{B}\overline{C} + \overline{A}BD$

(d) $(\overline{A} + \overline{B}) + (C + \overline{D})(B + \overline{C}) = \overline{A} + \overline{B} + BC + B\overline{D} + \overline{C}\overline{D}$
$= \overline{A}\overline{B} + C + \overline{D}$

(e) $\overline{A} + \overline{B}\overline{C} + CD$

3.13 The important columns in these tables are as follows:

X	Y	Z	$(\overline{X + Y + Z})$	$\overline{X}\overline{Y}\overline{Z}$
0	0	0	1	1
0	0	1	0	0
0	1	0	0	0
0	1	1	0	0
1	0	0	0	0
1	0	1	0	0
1	1	0	0	0
1	1	1	0	0

X	Y	Z	$(\overline{XYZ})$	$\overline{X} + \overline{Y} + \overline{Z}$
0	0	0	1	1
0	0	1	1	1
0	1	0	1	1
0	1	1	1	1
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	0	0

3.15 (a) $\overline{A}BC + A\overline{B}C + AC + BC = AC + BC$

(b) $A\overline{B} + A\overline{C} + ABC + AB\overline{C} + \overline{A}BC + \overline{B}C$ (This can be simplified to $\overline{B}C$)

(c) $A\overline{B} + A\overline{B}C = A\overline{B}$

3.17 Rule 13

3.19 (a) $AB + AD + BC + CD$

(b) $AB + AD + AC + BC + CD + C + BD + D + DC = C + D + AB$

(c) $ABC + ABD + AC + BC + DC + ADC + BDC + DC = AC + BC + DC + ABD$

(d) $A\overline{B} + A\overline{C}$

3.21 $\overline{X}\overline{Y}Z + XY\overline{Z}$

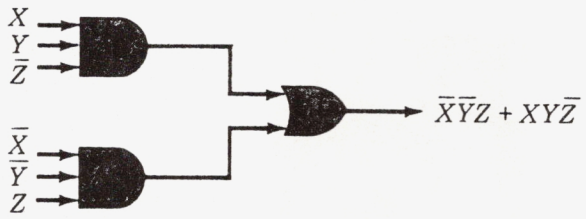


FIGURE A3.21

3.25 $XY + X\overline{Z}$

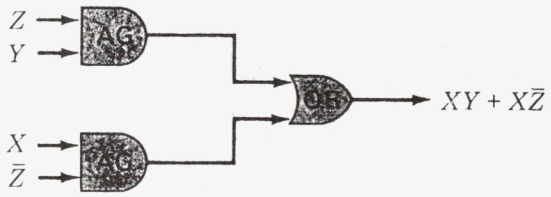


FIGURE A3.25

3.27

X	Y	$X + \overline{X}Y$	$X + Y$
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	1

3.29 $Y\overline{Z} + \overline{Y}Z$ is the sum-of-products expression, and $(Y + Z)(\overline{Y} + \overline{Z})$ is the product-of-sums expression.

3.35 (*a*)

	$\overline{X}$	X
$\overline{Y}$	1	
Y	1	

$\overline{X}\overline{Y} + \overline{X}Y$

(*b*)

	$\overline{X}\overline{Y}$	$\overline{X}Y$	XY	$X\overline{Y}$
$\overline{Z}$	1		1	
Z				

$\overline{X}\overline{Y}\overline{Z} + XY\overline{Z}$

(*c*)

	$\overline{X}\overline{Y}$	$\overline{X}Y$	XY	$X\overline{Y}$
$\overline{Z}$	1	1		
Z	1			

$\overline{X}Y\overline{Z} + \overline{X}\overline{Y}$

(*d*)

	$\overline{X}\overline{Y}$	$\overline{X}Y$	XY	$X\overline{Y}$
$\overline{Z}$	1	1		1
Z				

$\overline{X}\overline{Y}\overline{Z} + \overline{X}Y\overline{Z} + X\overline{Y}\overline{Z}$

(*e*)

	$\overline{X}\overline{Y}$	XY	XY	$X\overline{Y}$
$\overline{Z}$	1	1		1
Z				

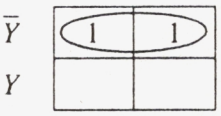
$\overline{X}\overline{Z} + \overline{Y}\overline{Z}$

(*f*)

	$\overline{A}\overline{B}$	$\overline{A}B$	AB	$A\overline{B}$
C				
C				

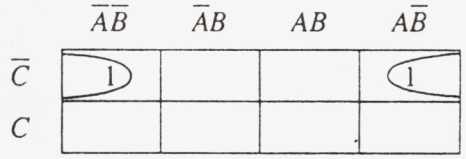
$AB(\overline{A}\overline{B}\overline{C} + \overline{B}C)$

3.39 (*a*)



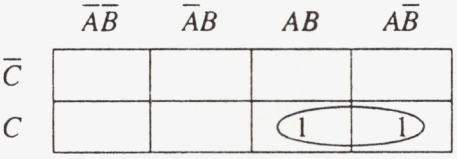
$X\overline{Y} + \overline{X}\overline{Y}$

(*b*)



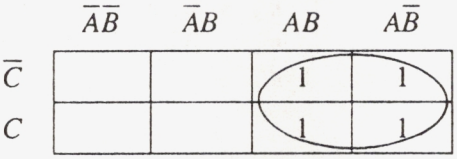
$\overline{A}\overline{B}\overline{C} + A\overline{B}\overline{C}$

(*c*)



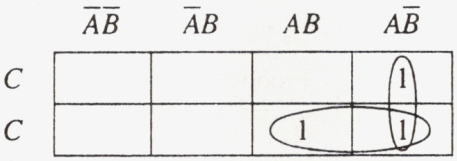
$A\overline{B}C + ABC$

(*d*)



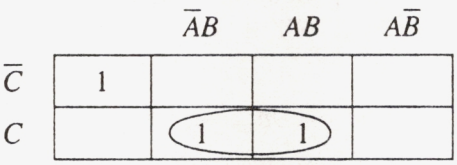
$ABC + A\overline{B}\overline{C} + A\overline{B}C + AB\overline{C}$

(*e*)



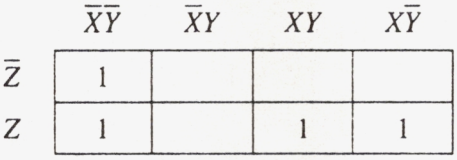
$ABC + A\overline{B}\overline{C} + A\overline{B}C$

(*f*)



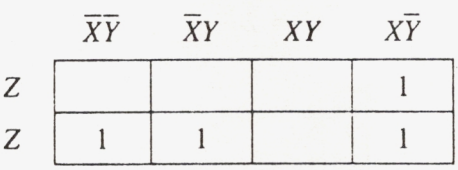
$ABC + \overline{A}BC + \overline{A}\overline{B}\overline{C}$

3.41 (*a*)



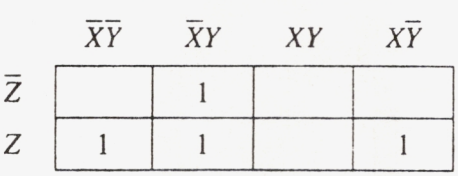
$m_0 + m_1 + m_5 + m_7$

(*b*)



$m_1 + m_3 + m_4 + m_5$

(*c*)



$m_1 + m_2 + m_3 + m_5$

(*d*)

	$\bar{X}\bar{Y}$	$\bar{X}Y$	XY	$X\bar{Y}$
$\bar{Z}$	1			
Z			1	1

$m_0 + m_5 + m_7$

3.45 (*b*)

	$\bar{W}\bar{X}$	$\bar{W}X$	WX	$W\bar{X}$
$\bar{Y}\bar{Z}$	1	1	1	1
$\bar{Y}Z$			1	1
YZ				1
$Y\bar{Z}$	1			1

$m_0 = \bar{W}\bar{X}\bar{Y}\bar{Z}$ $\quad m_{10} = W\bar{X}Y\bar{Z}$
$m_2 = \bar{W}\bar{X}Y\bar{Z}$ $\quad m_{11} = W\bar{X}YZ$
$m_4 = \bar{W}X\bar{Y}\bar{Z}$ $\quad m_{12} = W\bar{W}\bar{Y}\bar{Z}$
$m_8 = W\bar{X}\bar{Y}\bar{Z}$ $\quad m_{13} = WX\bar{Y}Z$
$m_9 = W\bar{X}\bar{Y}Z$

$m_0 + m_2 + m_4 + m_8 + m_9 + m_{10} + m_{11} + m_{12} + m_{13}$
$\bar{X}\bar{Z} + \bar{Y}\bar{Z} + W\bar{X} + W\bar{Y}$
$\bar{Z}(\bar{X} + \bar{Y}) + W(\bar{X} + \bar{Y})$
$(\bar{Z} + W)(\bar{X} + \bar{Y})$

3.47 (*a*) $m_1 + m_3 + m_5 + m_7 + m_{12} + m_{13} + m_8 + m_9$

	WX	$W\bar{X}$	$\bar{W}\bar{X}$	$\bar{W}X$
YZ	0	0	m_3	m_7
$Y\bar{Z}$	0	0	0	0
$\bar{Y}\bar{Z}$	m_{12}	m_8	0	0
$\bar{Y}Z$	m_{13}	m_0	m_1	m_5

Product of sums:

$\overline{(WY + \bar{W}Z)} = (\bar{W} + \bar{Y})(W + Z)$

Sum of products:

$W\bar{Y} + \bar{W}Z$

(*b*) $m_0 + m_5 + m_7 + m_8 + m_{11} + m_{13} + m_{15}$

	WX	$W\bar{X}$	$\bar{W}\bar{X}$	$\bar{W}X$
YZ	m_{15}	m_{11}		m_7
$Y\bar{Z}$	0			0
$\bar{Y}\bar{Z}$	0	m_8	m_0	0
$\bar{Y}Z$	m_{13}			m_5

Product of sums:

$(X\bar{Z} + Y\bar{Z} + \bar{W}\bar{X}Z + \bar{X}\bar{Y}Z)$
$= (\bar{X} + Z)(\bar{Y} + Z)(W + X + \bar{Z})$
$(X + Y + \bar{Z})$

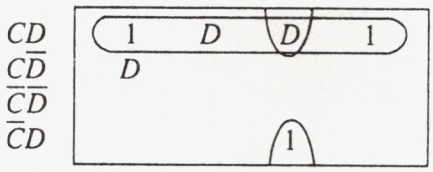
3.49 (*a*) $\overbrace{\bar{A}\bar{B}\bar{C} + A\bar{B}\bar{C} + ABC + \bar{A}B\bar{C} + \bar{A}\bar{B}C}^{\text{don't-cares}} = \bar{B}\bar{C}$
$\quad AB \quad A\bar{B} \quad \bar{A}\bar{B} \quad \bar{A}B$

C	D		D
C		1	1 $\quad D$

(*b*) $ABC + A\bar{B}\bar{C} + \overbrace{AB\bar{C} + A\bar{B}C}^{\text{don't-cares}} = A$
$\quad AB \quad A\bar{B} \quad \bar{A}\bar{B} \quad \bar{A}B$

C	1	D
$\bar{C}$	D	1

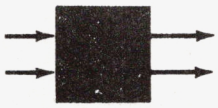
(c) $ABCD + \overline{A}\overline{B}\overline{C}D + \overline{A}BCD + \overbrace{A\overline{B}CD + \overline{A}\overline{B}CD + ABC\overline{D}}^{\text{don't-cares}}$



$CD + \overline{A}\overline{B}D$

Chapter 4

4.1



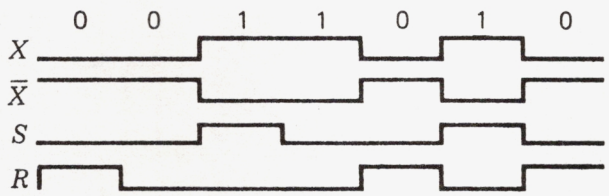
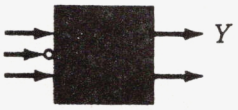


FIGURE A4.1

4.5



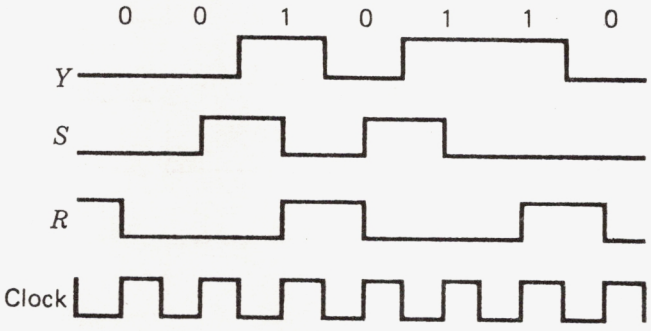


FIGURE A4.5

4.9

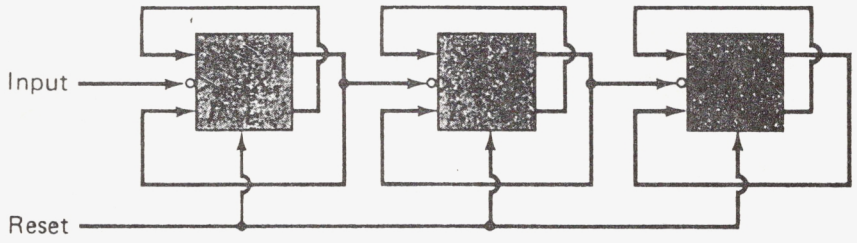
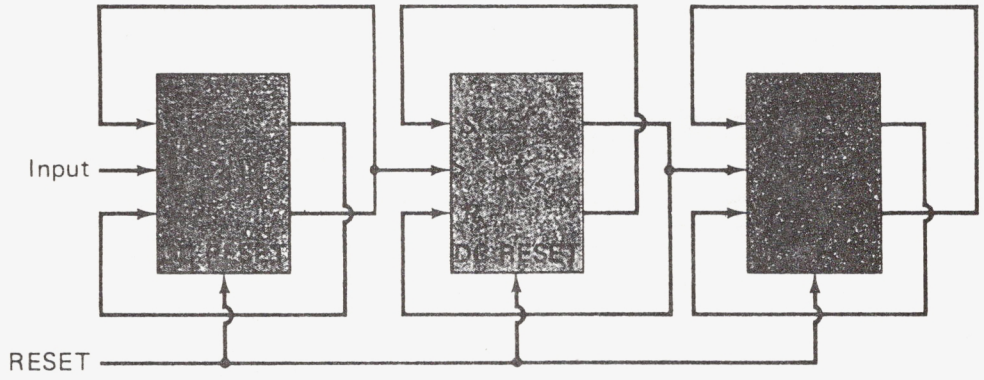


FIGURE A4.9

4.13



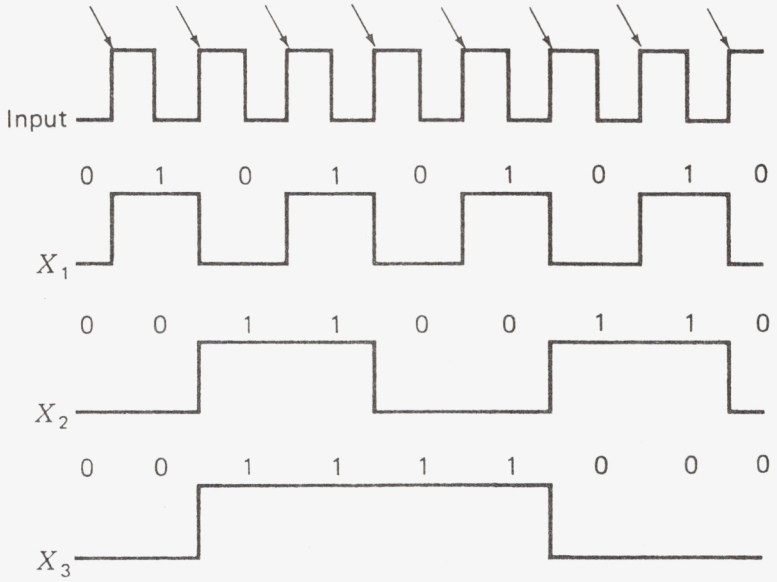


FIGURE A4.13

596

ANSWERS TO SELECTED ODD-NUMBERED QUESTIONS

4.15

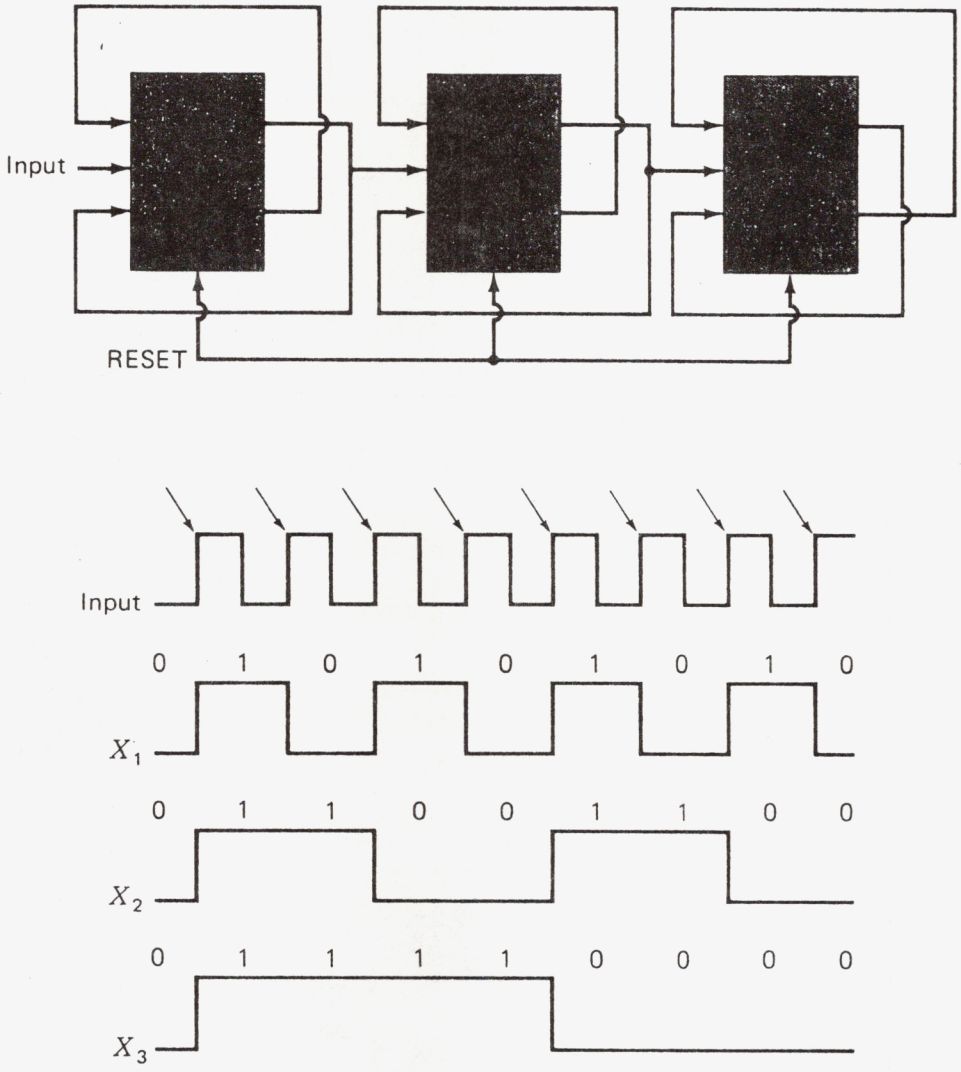


FIGURE A4.15

4.17

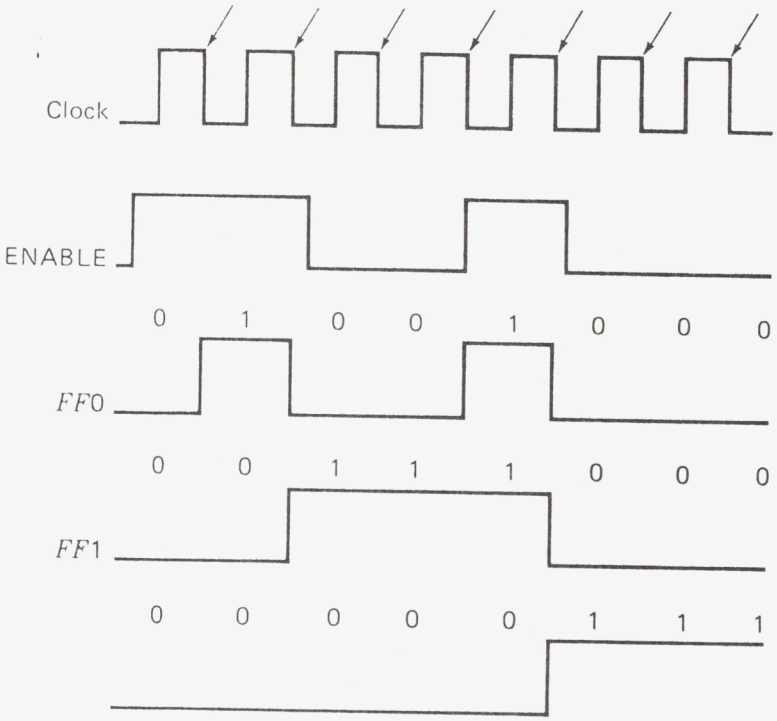


FIGURE A4.17

Chapter 5

5.1

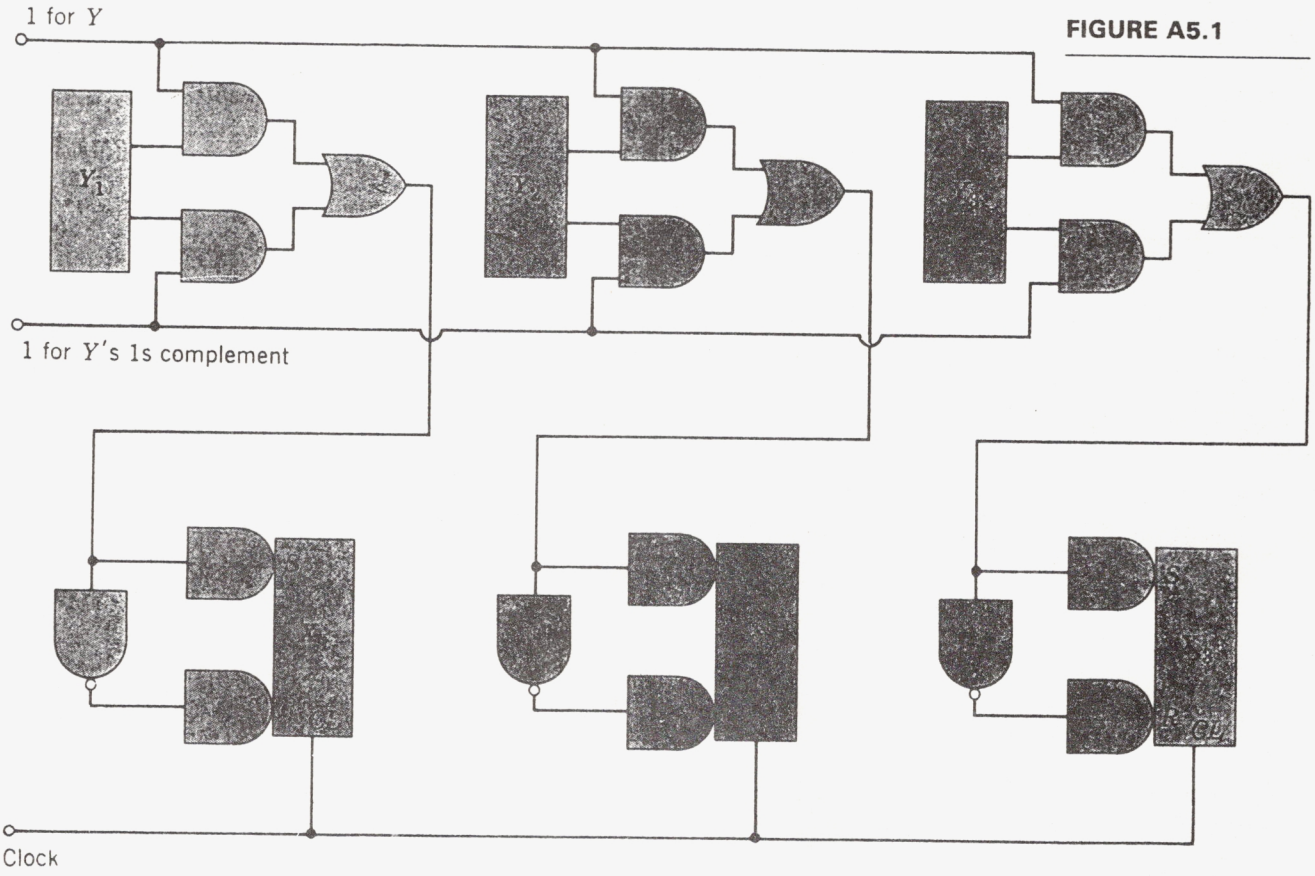


FIGURE A5.1

5.3 (*a*) 00110 (*b*) 01010 (*c*) 11100 (*d*) Not representable

5.5 −4 would be stored 10100 in the magnitude system, 11011 in the 1s complement system, and 11100 in the 2s complement system.

5.7 $S = 1$ and $C = 1$

5.9 −12 in 1s complement; −13 in 2s complement

5.11 The sum will overflow the register and cause an incorrect addition. Most machines sense for this and turn on an "addition overflow" or indicate the overflow in some manner.

5.15

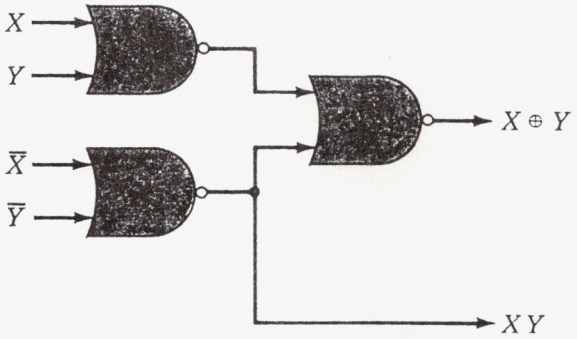


FIGURE A5.15

5.29 Logical addition, 0111011
Logical multiplication, 01000 10
Exclusive OR, 0011001

5.31 The logical MULTIPLY will clear those digits of X where 0s occur in Y, and the logical ADD will add 1s into the places in X where 1s occur in Y.

5.48

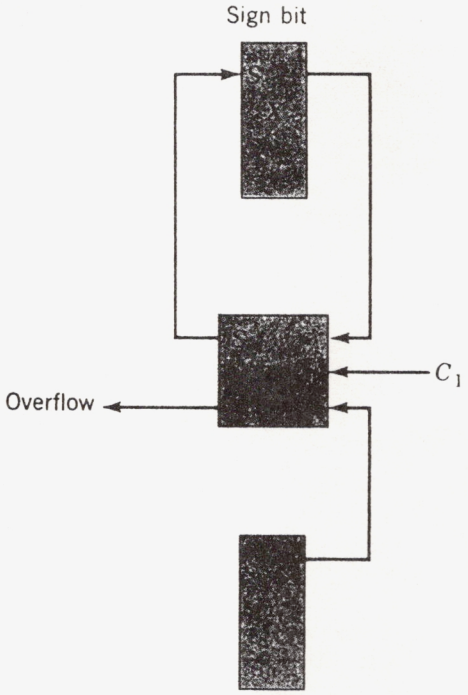


FIGURE A5.48

Overflow $= (X > \emptyset)\cdot(Y > \emptyset)\cdot(C_3 = 1) + (X < \emptyset)\cdot(Y < 0)\cdot(C_3 = \emptyset)$
Overflow $= \overline{X}_4\overline{Y}_4C_3 + X_4Y_4\overline{C}_3$

5.50 The general scheme is

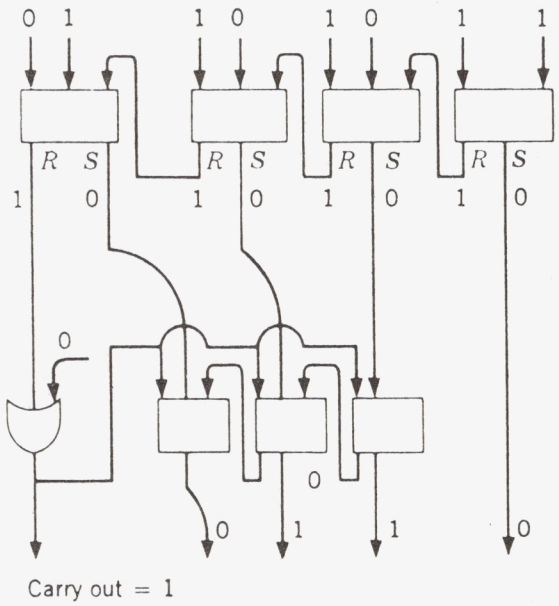


FIGURE A5.50

Chapter 6

6.12

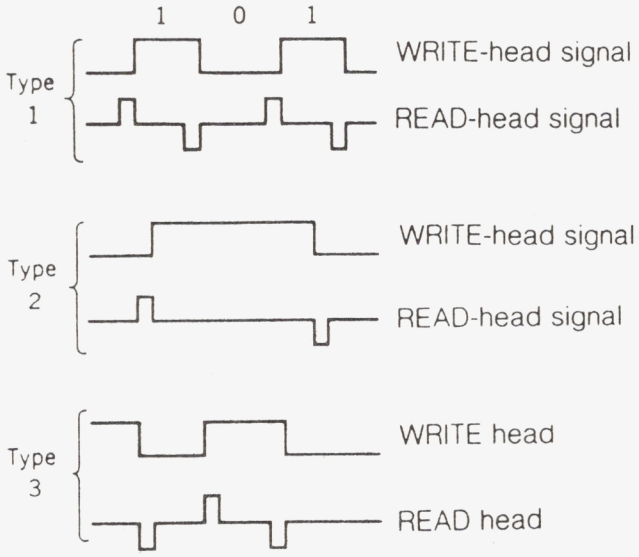


FIGURE A6.12

6.35 262,500 bits/s

Chapter 7

7.3 01100001
01100100
01100100
00010000
00010110
00000100
00000001
10000000 Carriage return is optional

7.5 27 holes in three cards. Easier to correct if cards are used, for erroneous cards may simply be replaced.

7.7 Nine holes

7.9

DECIMAL	EXCESS-3 CODE	EVEN-PARITY CHECK	ODD-PARITY CHECK
0	0011	0	1
1	0100	1	0
2	0101	0	1
3	0110	0	1
4	0111	1	0
5	1000	1	0
6	1001	0	1
7	1010	0	1
8	1011	1	0
9	1100	0	1

7.11 (*a*) The errors are in the seventh, eighth, and ninth rows; the sixth digit in each row is in error, and the message is, "That's right."
(*b*) The error is the fourth digit in the sixth row, and the message is, "Don't stop."

7.13 10. One such code can be formed by adding a leading 0 to each of the four-binary-digit Gray code groups listed in Chap. 8 and then adding another 16 rows with the four rightmost binary digits the same as those in Chap. 8 but with their order reversed and with a leading 1 added to each code row.

7.15 (*a*) Errors are in the second, third, and fourth rows, the seventh digit in. The message is, "That's fine."
(*b*) Errors are in the second and third rows, the third digit in, and the message is, "Bad tapes." Notice that we corrected a double error in a column.

Chapter 8

8.16 To interface a keyboard with device address 6, modify Fig. 8.20 as follows. The keyboard-to-bus data line drivers remain the same and are not shown here. The address line selection logic is changed:

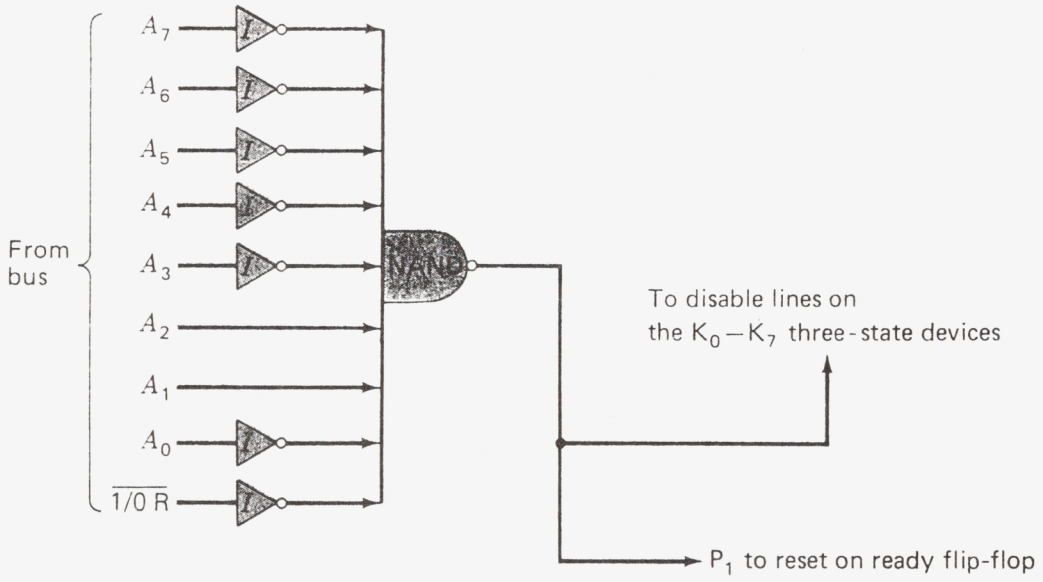


FIGURE A8.16

Chapter 9

9.3 The timing table is shown in Table 9.3 with the following additions:
OPCODE 00100 = BRM

BRM (from Table 9.4)

I and T_0	Set R	Tell memory to read instruction word
I and T_1	MB into OP,	Operation portion of instruction word into the operation register.
	RESET R	Clear memory read flip-flop.
I and T_2 and AC_0	MB into IC	If sign bit of accumulator is 1 (contents as negative), then use the address portion of the instruction address as the next instruction address.
I and T_2 and AC_0	INCREMENT IC	If sign bit of accumulator is 0 (contents are not negative), then use the next sequential address in memory as the next instruction address
I and T_3	IC INTO MA	Set up to read the next instruction from memory. Leave I = 1 so that the next cycle will be another instruction cycle.

BRA

I and T_0	SET R	Tell memory to read instruction
I and T_1	MB INTO OP,	Operation portion of instruction word into the operation register.
	RESET R	Clear memory read flip-flop.
I and T_2	MB INTO IC	Use address portion of instruction word as the next instruction address.
I and T_3	IC INTO MA	Set up to read the next instruction from memory. Leave I = 1 so that the next cycle will be another instruction cycle.

The control signal generation logic in Figs. 9.6 and 9.8 should be augmented with the following:

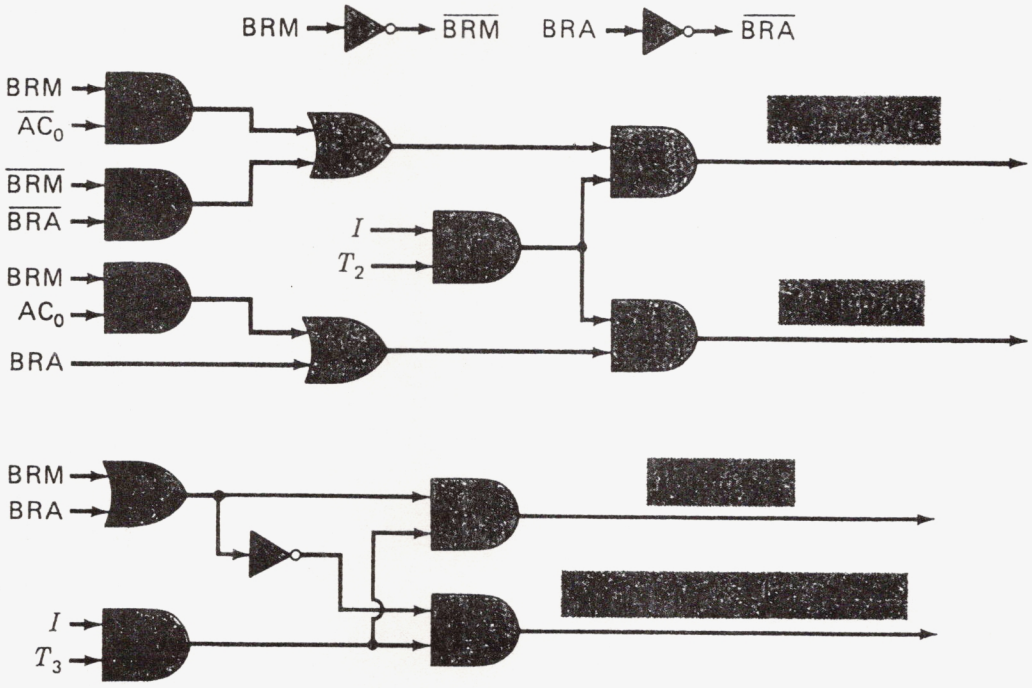


FIGURE A9.3

9.8 Instructions which require an execution cycle generally must access memory to obtain or to store an operand. Examples from the instruction set of Table 9.3 include ADD and STORE. Instructions which do not require an access to memory for an operand generally do not require an execution cycle. In such cases there may be no operand; or the operand has been obtained as part of the instruction and is manipulated solely within the central registers of the processor. Examples are BRM, BRA, and CLR.

9.18 This question says: Provide a microprogram for the BRM branch-on-minus instruction of Table 9.4. The OP code in the BRM instruction is 00100 = 4_{10}.

MICROPROGRAM ROM LOCATION	C_0 to C_6	MICROPROGRAM C_7 to C_n
0	—	$1 \rightarrow R$; IAR + $1 \rightarrow$ IAR
1	—	$MB_{0-4} \rightarrow OP$; IAR + $1 \rightarrow$ IAR; $0 \rightarrow R$
2	—	OP + IAR + $1 \rightarrow$ IAR
3	20_{10}	$C_{0-6} \rightarrow$ IAR
.	.	.
.	.	.
.	.	.
7	40_{10}	$C_{0-6} \rightarrow$ IAR
.	.	.
.	.	.
.	.	.
40	42_{10}	IF (AC_0 = 1) THEN $C_{0-6} \rightarrow$ IAR; ELSE IAR + $1 \rightarrow$ IAR
41	43_{10}	$C_{0-6} \rightarrow$ IAR; IC + $1 \rightarrow$ IC
42	—	$MB_{0-24} \rightarrow$ IC
43	—	IC $\rightarrow$ MA; $0 \rightarrow$ IAR
.	.	.
.	.	.
.	.	.

2 8086 Architecture

Copied pursuant to S53B of the Copyright Act as amended on behalf of Murdoch Uni. on 17.2.89

Liu, Yu-Cheng, and Gibson, Glenn A., Microcomputer Systems : The 8086/8088 family, Prentice Hall, 1984 pp.ix + 550, ISBN 0 13 580944 4
pp. 25-52, 55-63, 128-134, 536-543

The Intel 8086, a 16-bit microprocessor, contains approximately 29,000 transistors and is fabricated using the HMOS technology. Its throughput is a considerable improvement over that of the Intel 8080, its 8-bit predecessor. Although some attempt at compatibility with the 8080 CPU architecture was made, the designers decided not to sacrifice sophistication in order to attain compatibility (Morse et al. [1]). By increasing the number of address pins from 16 to 20, the memory addressing capacity was increased from 64K bytes to $2^{20} \approx 1$ megabyte. The expanded memory capability made multiprogramming feasible and several multiprogramming features have been incorporated into the 8086's design. The 8086 also includes a number of features which enhance its multiprocessing capabilities, thus allowing it to be used with other processing elements such as the 8087 numeric data processor.

A pin assignment diagram for the 8086 is given in Fig. 2-1. The 8086 has 20 address pins, 16 of which are also used as data pins. The use of pins for both addresses and data means that both an address and datum cannot be sent to the system bus at the same time. This multiplexing of addresses and data reduces the number of pins needed, but does slow down the transfer of data. However, because of the timing on the bus, the transfer rate is not decreased as much as one might expect. There are 16 control lines for providing handshaking signals during bus transfers and for permitting at least some external control of the CPU. The 8086 requires only one supply voltage, +5 V, and one clock phase whose frequency can be up to 5 MHz. (There are actually two other versions of the 8086, the 8086-2, which permits a clock frequency of up to 8 MHz, and the 8086-1, which can handle up to 10 MHz.) Rounding out the 40-pin configuration are two grounds, pins 1 and 20.

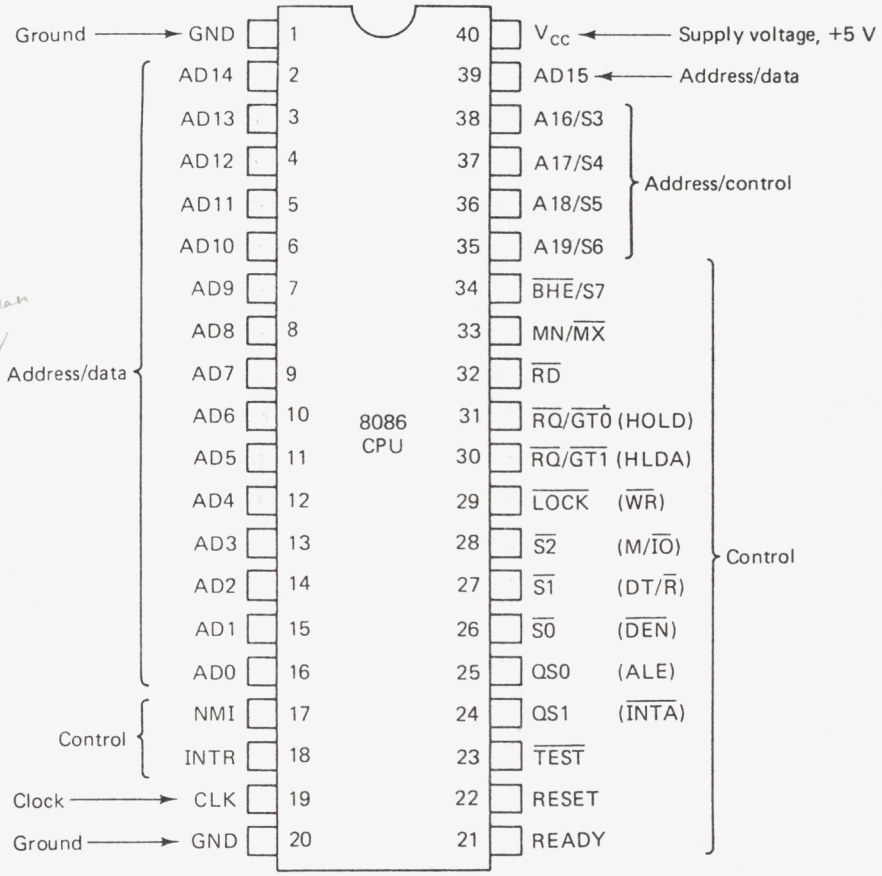
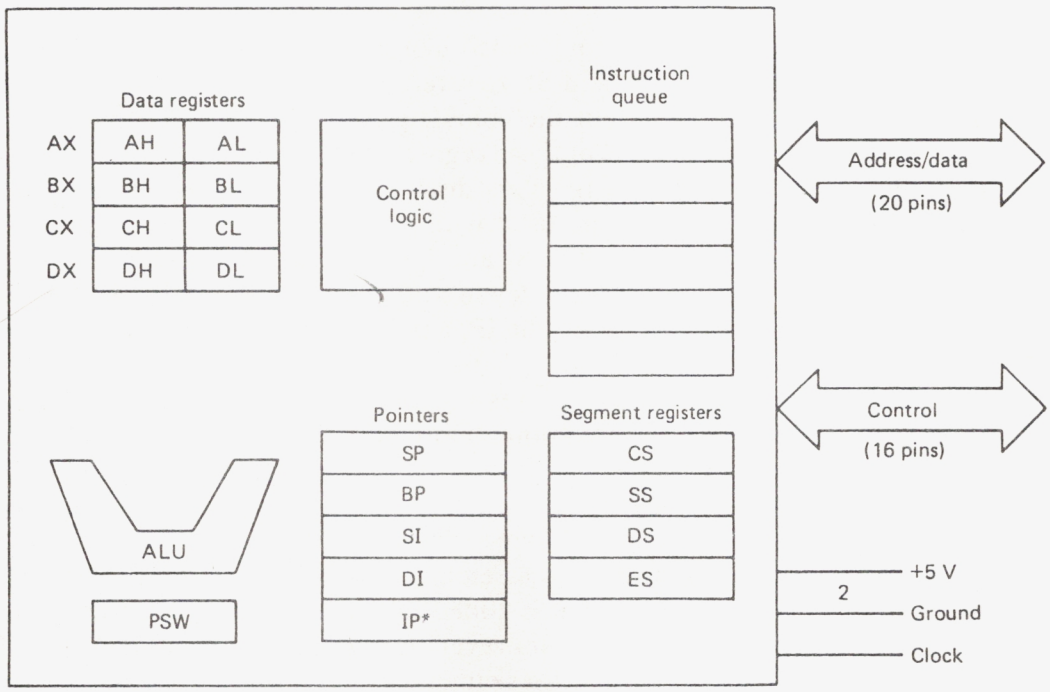


Figure 2-1 8086 pin assignments. (Reprinted by permission of Intel Corporation. Copyright 1981.)

This chapter begins with a description of the 8086's CPU architecture and internal operation. The third section defines the ways in which the 8086 can address data and perform branches and discusses the 8086's machine language instruction formats. Section 2-4 discusses the amounts of time needed to execute the various instructions and Sec. 2-5 describes the Intel 8088, which is a software compatible 8-bit version of the 8086. The 8088 was designed to allow 8080 and 8085 hardware configurations to be upgraded to an 8086 software environment.

2-1 CPU ARCHITECTURE

Figure 2-2 shows the internal architecture of the 8086. Except for the instruction register, which is actually a 6-byte queue, the control unit and working registers described in Sec. 1-4 are divided into three groups according to their functions. There are the data group, which is essentially the set of arithmetic registers; the pointer group, which includes base and index registers, but also contains the pro-



*For the 8086 the program counter is called the instruction pointer (IP).

Figure 2-2 8086's internal configuration.

gram counter and stack pointer; and the segment group, which is a set of special purpose base registers. All of the registers are 16 bits wide.

The data group consists of the AX, BX, CX, and DX registers. These registers can be used to store both operands and results and each of them can be accessed as a whole, or the upper and lower bytes can be accessed separately. For example, either the 2 bytes in AX can be used together, or the upper byte AH or the lower byte AL can be used by itself by specifying AH or AL, respectively. For purposes of converting 8080 software into 8086 software the following correspondences can be drawn:

8086	8080
AL	A
BH	H
BL	L
CH	B
CL	C
DH	D
DL	E

In addition to serving as arithmetic registers, the BX, CX, and DX registers play special addressing, counting, and I/O roles:

BX may be used as a base register in address calculations.

CX is used as an implied counter by certain instructions.

DX is used to hold the I/O address during certain I/O operations.

The pointer and index group consists of the IP, SP, BP, SI, and DI registers. The instruction pointer (IP) and SP registers are essentially the program counter and stack pointer registers, but the complete instruction and stack addresses are formed by adding the contents of these registers to the contents of the code segment (CS) and stack segment (SS) registers discussed below. BP is a base register for accessing the stack and may be used with other registers and/or a displacement that is part of the instruction. The SI and DI registers are for indexing. Although they may be used by themselves, they are often used with the BX or BP registers and/or a displacement. Except for the IP, a pointer can be used to hold an operand, but must be accessed as a whole.

To provide flexible base addressing and indexing, a data address may be formed by adding together a combination of the BX or BP register contents, SI or DI register contents, and a displacement. The result of such an address computation is called an *effective address* (EA) or *offset*. [The Intel manuals tend to use the term "effective address" when discussing the machine language and the term "offset" when discussing the assembler language. The word "displacement" is used to indicate a quantity that is added to the contents of a register(s) to form an EA.] The final data address, however, is determined by the EA and the appropriate data segment (DS), extra segment (ES), or stack segment (SS) register.

The segment group consists of the CS, SS, DS, and ES registers. As indicated above, the registers that can be used for addressing, the BX, IP, SP, BP, SI, and DI registers, are only 16 bits wide and, therefore, an effective address has only 16 bits. On the other hand, the address put on the address bus, called the *physical address*, must contain 20 bits. The extra 4 bits are obtained by adding the effective address to the contents of one of the segment registers as shown in Fig. 2-3. The addition is carried out by appending four 0 bits to the right of the number in the segment register before the addition is made; thus a 20-bit result is produced. As an example, if (CS) = 123A and (IP) = 341B, then the next instruction will be fetched from

341B	Effective address
+ 123A0	Beginning segment address
157BB	Physical address of instruction

[It is standard notation for parentheses around an entity to mean "contents of," e.g., (IP) means the contents of IP. Also, all addresses are given in hexadecimal.]

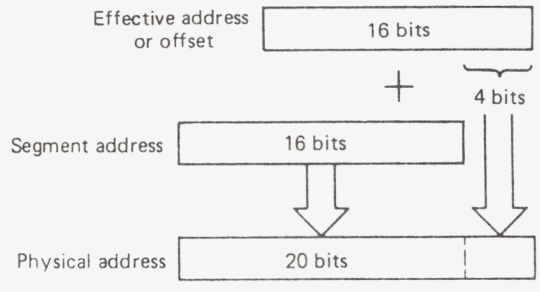


Figure 2-3 Formation of a physical address.

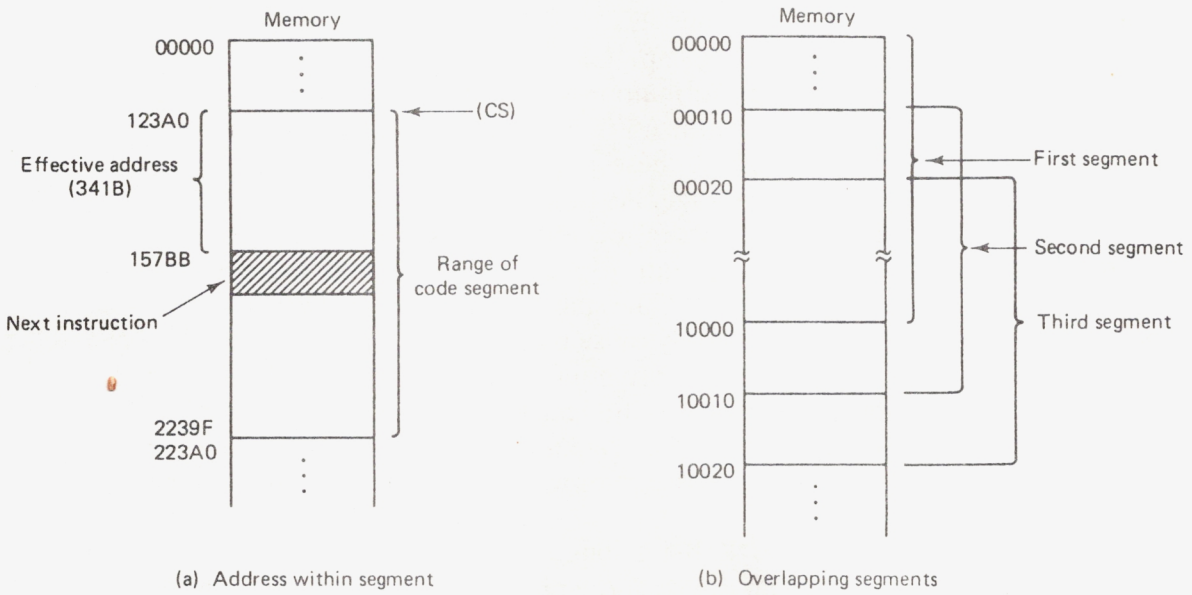
The utilization of the segment registers essentially divides the memory space into overlapping segments, with each segment being 64K bytes long and beginning at a 16-byte, or paragraph, boundary, i.e., beginning at an address that is divisible by 16. We will hereafter refer to the contents of a segment register as the *segment address*, and the segment address multiplied by 16_{10} as the *beginning physical segment address*, or simply, the *beginning segment address*. An illustration of the example above is given in Fig. 2-4(a) and the overall segmentation of memory is shown in Fig. 2-4(b).

The advantages of using segment registers are that they:

1. Allow the memory capacity to be 1 megabyte even though the addresses associated with the individual instructions are only 16 bits wide.
2. Allow the instruction, data, or stack portion of a program to be more than 64K bytes long by using more than one code, data, or stack segment.
3. Facilitate the use of separate memory areas for a program, its data, and the stack.
4. Permit a program and/or its data to be put into different areas of memory each time the program is executed.

Figure 2-5 shows how a program's code and its associated data and stack can be separated in memory. The simpler and conventional approach is to let both the code and the data reside in one contiguous area in memory and put the stack in some fixed area which always begins at, say, address 08000. This is satisfactory if there is only one program in memory at a time, but in a multiprogramming environment there may be several programs in memory simultaneously. For multi-

Figure 2-4 Address computations and memory segmentation.



(a) Address within segment

(b) Overlapping segments

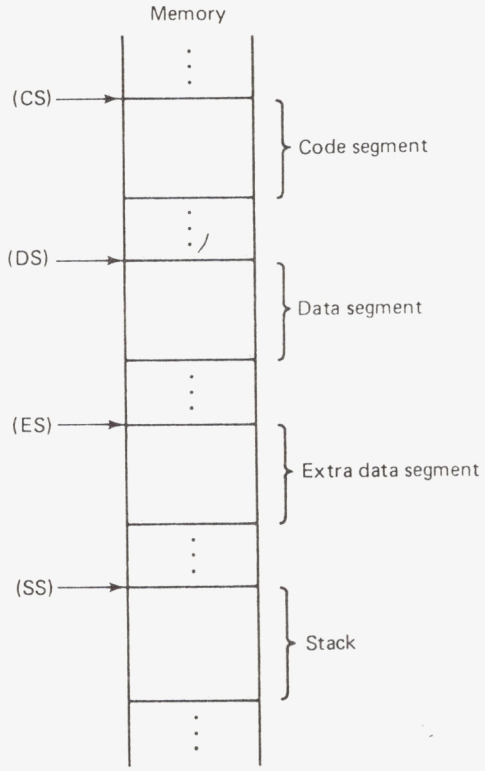


Figure 2-5 Separation of a program's code, its data, and its stack.

programming it is better to keep the contiguous chunks of memory as small as possible and, therefore, it is better to separate the code from the data. Analogously, the space in a pail is more fully utilized by filling it with small stones than with large stones. The option of making the code and data contiguous is always present, because the contents of CS, DS, and ES can always be made equal. Multiprogramming is examined in detail in Chap. 7.

Advantage 4 is also related to multiprogramming. For example, consider a time-sharing system in which several users are executing different programs. If there is not enough memory space, these programs are shuttled back and forth between the memory and mass storage. As they take turns using memory they are dynamically put into different places in memory. A given program may begin at physical address 02000 while it is currently executing, but the next time it is brought into memory it may begin at 1A300. The question is: If the program contains a branch instruction that would currently cause a branch to 02600, would the branch be correctly taken to 1A900 when the program is relocated to 1A300? The answer is yes provided that the contents of CS are changed to 1A300 when the program is relocated. This situation is depicted in Fig. 2-6. Similar statements can be made regarding the relocation of a block of data or a stack. Relocation will be discussed more fully in Chap. 7.

Although segments are 64K bytes long, they can be overlapped to better utilize the memory. If, as shown in Fig. 2-7, a program fills only part of a segment, another code, data, or stack segment could overlap the program's segment and the

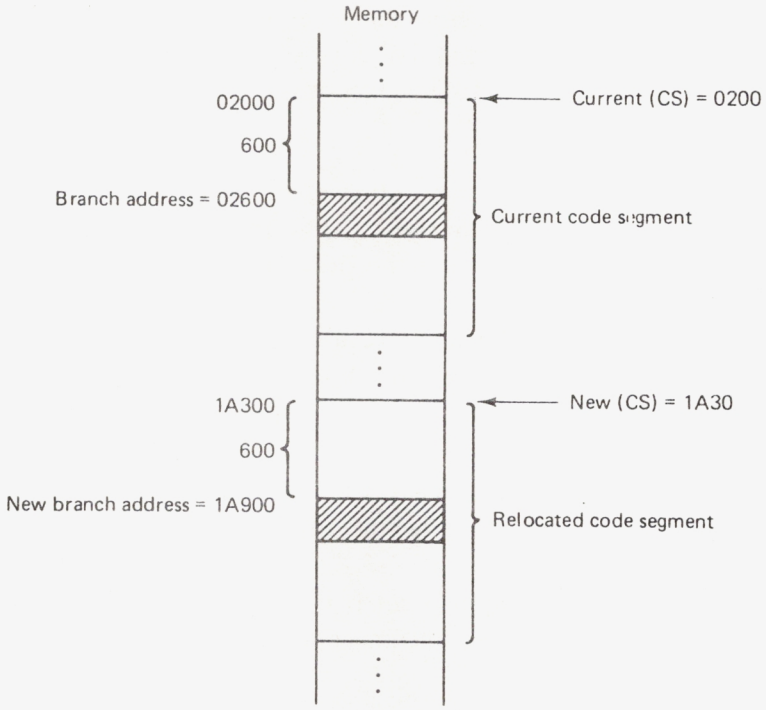


Figure 2-6 Program relocation using the CS register.

beginning address of the second segment could be within 16 bytes of the end of the program. The only wasted space would be the few bytes between the end of the program and the next 16-byte boundary.

The 8086's PSW contains 16 bits, but 7 of them are not used. Each bit in the PSW is called a *flag*. The 8086 flags are divided into the *conditional flags*, which reflect the result of the previous operation involving the ALU, and the *control*

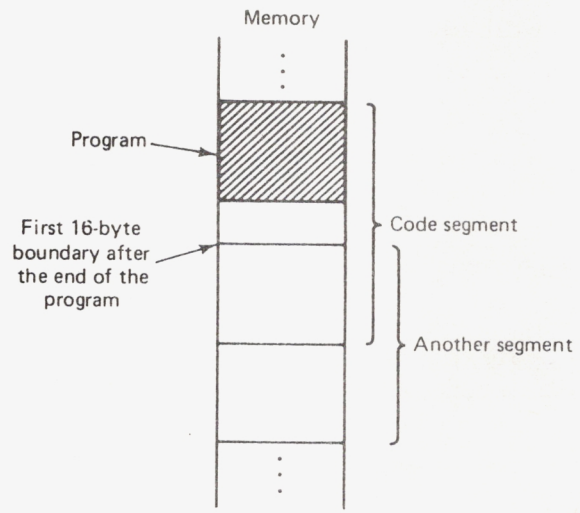


Figure 2-7 Overlapping segments.

flags, which control the execution of special functions. The flags are summarized in Fig. 2-8. The lower byte in the PSW corresponds to the 8-bit PSW in the 8080 and contains all of the condition flags except OF.

The condition flags are:

SF (Sign Flag)—Is equal to the MSB of the result. Since in 2's complement negative numbers have a 1 in the MSB and for nonnegative numbers this bit is 0, this flag indicates whether the previous result was negative or nonnegative.

ZF (Zero Flag)—Is set to 1 if the result is zero and 0 if the result is nonzero.

PF (Parity Flag)—Is set to 1 if the low-order 8 bits of the result contain an even number of 1s; otherwise it is cleared.

CF (Carry Flag)—An addition causes this flag to be set if there is a carry out of the MSB, and a subtraction causes it to be set if a borrow is needed. Other instructions also affect this flag and its value will be discussed when these instructions are defined.

AF (Auxiliary Carry Flag)—Is set if there is a carry out of bit 3 during an addition or a borrow by bit 3 during a subtraction. This flag is used exclusively for BCD arithmetic.

OF (Overflow Flag)—Is set if an overflow occurs, i.e., a result is out of range. More specifically, for addition this flag is set when there is a carry into the MSB and no carry out of the MSB or vice versa. For subtraction, it is set when the MSB needs a borrow and there is no borrow from the MSB, or vice versa.

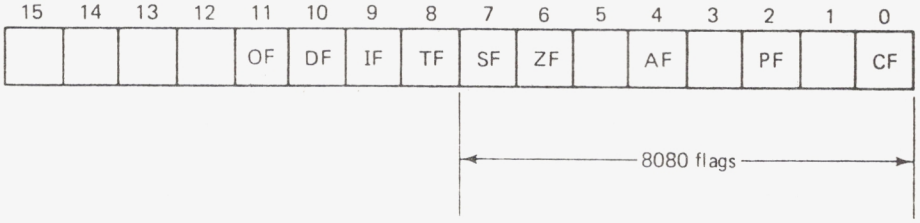
As an example, if the previous instruction performed the addition

$$\begin{array}{rr} & 0010\ 0011\ 0100\ 0101 \\ + & \underline{0011\ 0010\ 0001\ 1001} \\ & 0101\ 0101\ 0101\ 1110 \end{array}$$

then following the instruction:

SF = 0 ZF = 0 PF = 0 CF = 0 AF = 0 OF = 0

Figure 2-8 8086's PSW.



If the previous instruction performed the addition

$$\begin{array}{rr} & 0101\ 0100\ 0011\ 1001 \\ + & 0100\ 0101\ 0110\ 1010 \\ \hline & 1001\ 1001\ 1010\ 0011 \end{array}$$

then the flags would be:

SF = 1 ZF = 0 PF = 1 CF = 0 AF = 1 OF = 1

The control flags are:

DF (Direction Flag)—Used by string manipulation instructions. If clear, the string is processed from its beginning with the first element having the lowest address. Otherwise, the string is processed from the high address towards the low address.

IF (Interrupt Enable Flag)—If set, a certain type of interrupt (a maskable interrupt) can be recognized by the CPU; otherwise, these interrupts are ignored. (See Chaps. 4 and 6.)

TF (Trap Flag)—If set, a trap is executed after each instruction. (See Chap. 4.)

2-2 INTERNAL OPERATION

The general operation of a computer as outlined in Sec. 1-4 consists of:

1. Fetching the next instruction from the address indicated by the PC.
2. Putting it in the instruction register and decoding it while the PC is incremented to point to the next instruction.
3. Executing the instruction and, if a branch is to be taken, resetting the PC to the branch address.
4. Repeating steps 1 through 3.

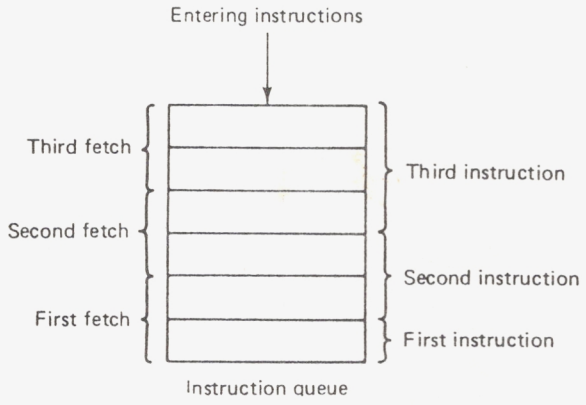
The operation of the 8086 follows this basic pattern, but there are differences and some of the operations may be overlapped.

The address of the next instruction is determined by the sum of the (IP) and (CS) $\times$ 16_{10}, and the instruction register is a 6-byte first-in/first-out (FIFO) queue that is continually being filled whenever the system bus is not needed for some other operation. This "look ahead" feature can significantly increase the CPU's throughput because much of the time the next instruction is already in the CPU when the present instruction completes its execution. If a branch is taken, then the instruction queue is flushed out and there is no time saving, but on the average this occurs only a small percent of the time. To demonstrate the time available for loading the queue let us consider an 8-bit multiply instruction. This instruction

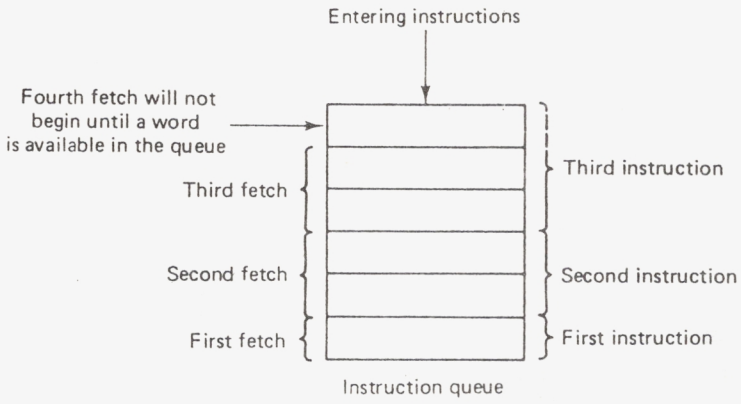
requires at least 71 clock cycles, but only 4 clock cycles are needed to input a word from memory. Therefore, during its execution there would be an ample number of extra clock cycles in which the bus is free to fill the instruction queue.

Although the 8086 can address a word which begins with either an odd address or an even address, for the odd-address case two memory references are required. One memory access is needed for the low-order byte and one for the high-order byte. This means that time is saved if words are stored at only even addresses. Even though the 8086's instructions may be from 1 to 6 bytes long, the 6-byte queue makes it possible to bring instructions in a word at a time using even addresses. The only exception occurs when there is a branch to an odd address. When this happens, the 8086 brings in 1 byte and then continues with even-address words. Figure 2-9(a) shows how the queue is filled by a sequence of the form:

1-byte instruction.
2-byte instruction.
3-byte instruction.



(a) First instruction has an even address



(b) First instruction has an odd address

Figure 2-9 Filling the instruction queue after a branch.

beginning with an even address, and Fig. 2-9(b) shows the same sequence but assumes an odd beginning address. Note that in the latter case the last byte of the third instruction will not be brought into the queue until an empty word in the queue becomes available.

2-3 MACHINE LANGUAGE INSTRUCTIONS

An instruction is divided into groups of bits, or *fields*, with one field, called the *operation code* (or *op code*), indicating what the computer is to do, and the other fields, called the *operands*, indicating the information needed by the instruction in carrying out its task. An operand may contain a datum, at least part of the address of a datum, an indirect pointer to a datum, or other information pertaining to the data to be acted on by the instruction. A general instruction format is shown in Fig. 2-10.

Instructions may contain several operands, but the more operands and the longer these operands are, the more memory space they will occupy and the more time it will take to transfer each instruction into the CPU. In order to minimize the total number of bits in an instruction, most instructions, particularly those for 16-bit computers, are limited to one or two operands with at least one operand in a two-operand instruction involving a register. Because the memory and/or I/O spaces are relatively large, memory and I/O addresses require several bits, but because the number of registers is small, it takes only a few bits to specify a register. Therefore, one means of conserving instruction bits is to use registers as much as possible. The two-operand limitation does reduce the flexibility of many instructions, but normally the extra flexibility is not really needed. For example, an addition instruction, which involves the two numbers being added and the result, is reduced to two operands by putting the sum into the location that contained the addend. This means that the addend is lost, but this is usually not important. If it is important, the addend could be duplicated (by also storing it elsewhere) before the addition is executed.

2-3-1 Addressing Modes

The way in which an operand is specified is called its *addressing mode*. The addressing modes for the 8086 instructions are typical and are discussed below. They are broken into two categories, those for data and those for branch addresses. Figure 2-11 graphically shows how the operands are determined for the various data-related addressing modes. These modes are:

Immediate—The datum is either 8 bits or 16 bits long and is part of the instruction.

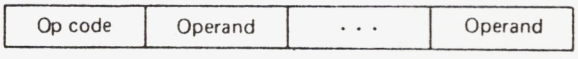
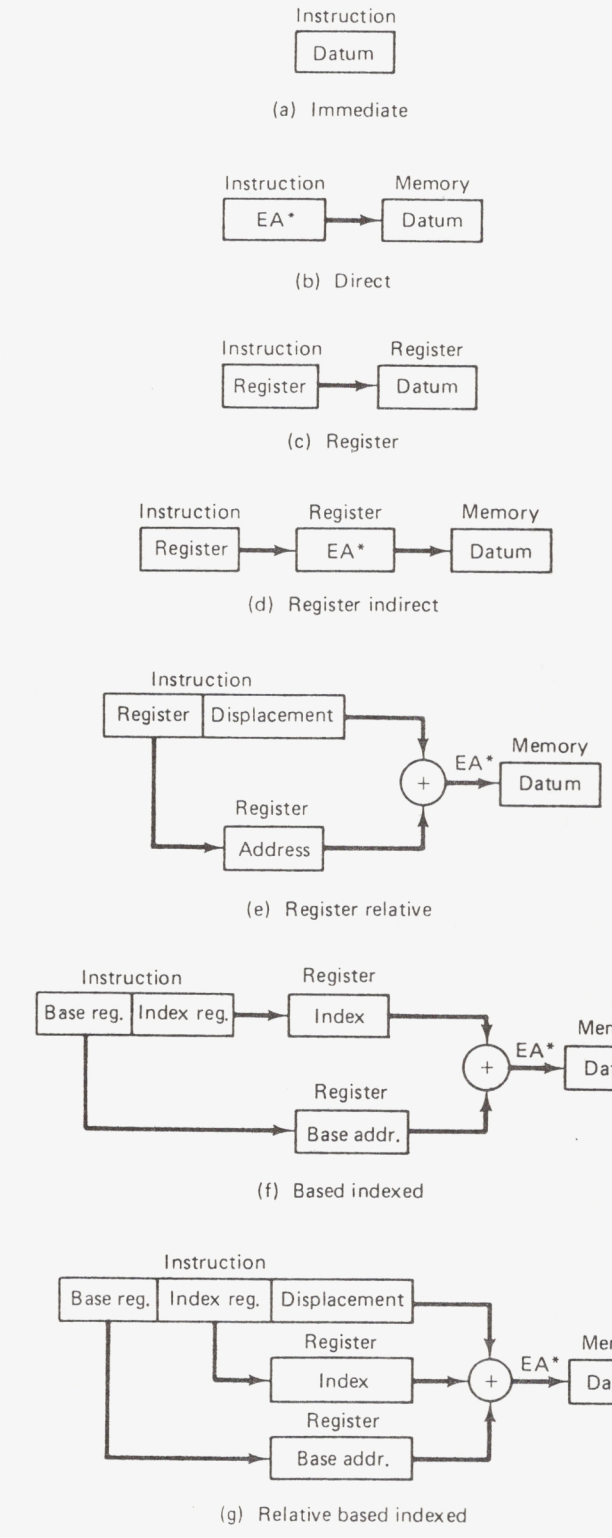


Figure 2-10 General instruction format.



*EA is added to 16_{10} times the contents of the appropriate segment register.

Figure 2-11 Data-related addressing modes.

Direct—The 16-bit effective address of the datum is part of the instruction.

Register—The datum is in the register that is specified by the instruction. For a 16-bit operand, a register may be AX, BX, CX, DX, SI, DI, SP, or BP, and for an 8-bit operand a register may be AL, AH, BL, BH, CL, CH, DL, or DH.

Register Indirect—The effective address of the datum is in the base register BX or an index register that is specified by the instruction, i.e.,

$$EA = \begin{Bmatrix} (BX) \\ (DI) \\ (SI) \end{Bmatrix}$$

Register Relative—The effective address is the sum of an 8- or 16-bit displacement and the contents of a base register or an index register, i.e.,

$$EA = \begin{Bmatrix} (BX) \\ (BP) \\ (SI) \\ (DI) \end{Bmatrix} + \begin{Bmatrix} \text{8-bit displacement} \\ \\ \text{16-bit displacement} \end{Bmatrix}$$

Based Indexed—The effective address is the sum of a base register and an index register, both of which are specified by the instruction, i.e.,

$$EA = \begin{Bmatrix} (BX) \\ (BP) \end{Bmatrix} + \begin{Bmatrix} (SI) \\ (DI) \end{Bmatrix}$$

Relative Based Indexed—The effective address is the sum of an 8- or 16-bit displacement and a based indexed address, i.e.,

$$EA = \begin{Bmatrix} (BX) \\ (BP) \end{Bmatrix} + \begin{Bmatrix} (SI) \\ (DI) \end{Bmatrix} + \begin{Bmatrix} \text{8-bit displacement} \\ \text{16-bit displacement} \end{Bmatrix}$$

For example, if

(BX) = 0158 (DI) = 10A5 Displacement = 1B57 (DS) = 2100

and DS is used as the segment register, then the effective and physical addresses produced by these quantities and the various addressing modes would be

Direct: EA = 1B57
Physical address = 1B57 + 21000 = 22B57

Register: No effective address—datum is in specified register.

Register indirect assuming register BX:
EA = 0158
Physical address = 0158 + 21000 = 21158

Register relative assuming register BX:
EA = 0158 + 1B57 = 1CAF
Physical address = 1CAF + 21000 = 22CAF

Based indexed assuming registers BX and DI:

EA = 0158 + 10A5 = 11FD

Physical address = 11FD + 21000 = 221FD

Relative based indexed assuming BX and DI:

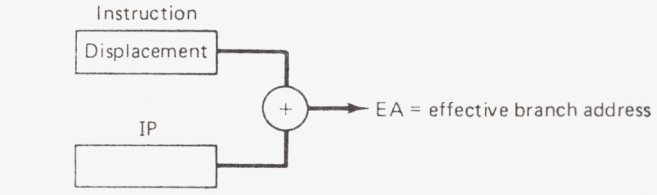
EA = 0158 + 10A5 + 1B57 = 2D54

Physical address = 2D54 + 21000 = 23D54

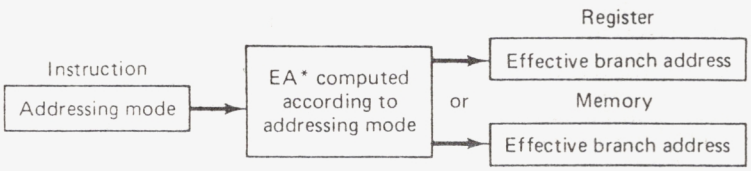
1B57
0158
10A5
2D54

The addressing modes for indicating branch addresses are graphically defined in Fig. 2-12 and are:

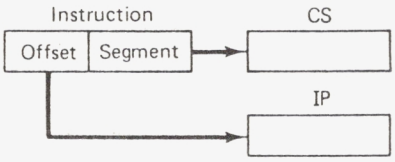
Intrasegment Direct—The effective branch address is the sum of an 8- or 16-bit displacement and the current contents of IP. When the displacement is 8 bits long, it is referred to as a short jump. Intrasegment direct addressing is what most computer books refer to as relative addressing because the dis-



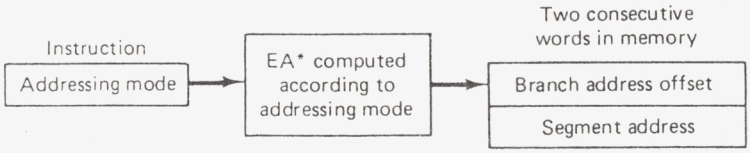
(a) Intrasegment direct



(b) Intrasegment indirect



(c) Intersegment direct



(d) Intersegment indirect

*EA is added to 16_{10} times the contents of the appropriate segment register.

Figure 2-12 Branch-related addressing modes.

placement is computed "relative" to the IP. It may be used with either conditional or unconditional branching, but a conditional branch instruction can have only an 8-bit displacement.

Intrasegment Indirect—The effective branch address is the contents of a register or memory location that is accessed using any of the above data-related addressing modes except the immediate mode. The contents of IP are replaced by the effective branch address. This addressing mode may be used only in unconditional branch instructions.

Intersegment Direct—Replaces the contents of IP with part of the instruction and the contents of CS with another part of the instruction. The purpose of this addressing mode is to provide a means of branching from one code segment to another.

Intersegment Indirect—Replaces the contents of IP and CS with the contents of two consecutive words in memory that are referenced using any of the above data-related addressing modes except the immediate and register modes.

Note that the physical branch address is the new contents of IP plus the contents of CS multiplied by 16_{10}. An intersegment branch must be unconditional.

To demonstrate how indirect branching works with some of the data-related addressing modes, suppose that

(BX) = 1256 (SI) = 528F Displacement = 20A1

Then:

With direct addressing, the effective branch address is the contents of:

$$20A1 + (DS) \times 16_{10}$$

With register relative addressing assuming register BX, the effective branch address is the contents of:

$$1256 + 20A1 + (DS) \times 16_{10}$$

With based indexed addressing assuming registers BX and SI, the effective branch address is the contents of:

$$1256 + 528F + (DS) \times 16_{10}$$

2-3-2 Instruction Formats

Several representative 8086 instruction formats are shown in Fig. 2-13. The instructions vary from 1 to 6 bytes in length and a complete summary of them is given in Sec. 3-12. Displacements and immediate data may be either 8 bits or 16 bits long depending on the instruction. The op code and addressing mode designations are in the first 1 or 2 bytes of an instruction. The op code/addressing mode

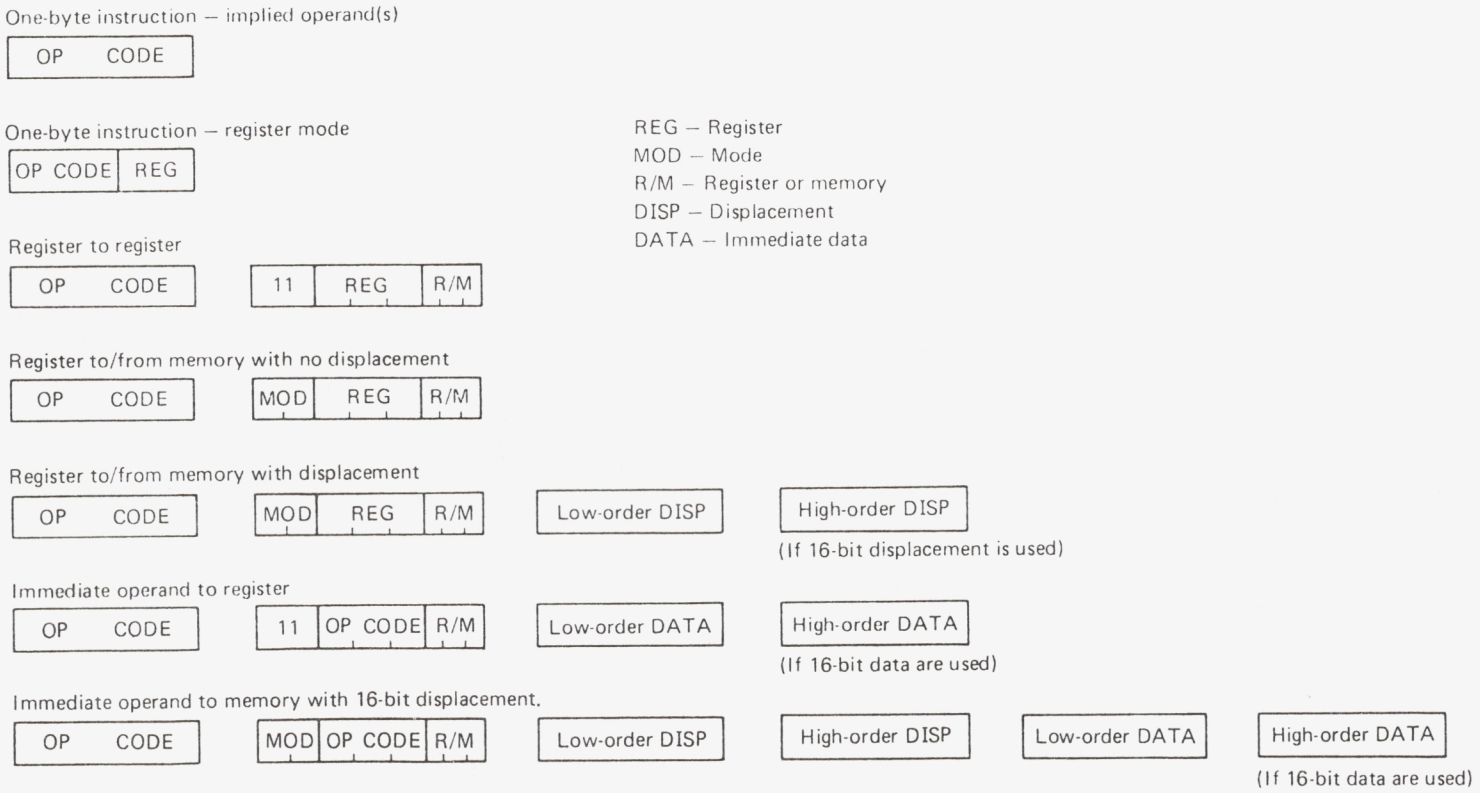


Figure 2-13 Examples of 8086 instruction formats.

byte(s) may be followed by:

No additional bytes.
A 2-byte EA (for direct addressing only.)
A 1- or 2-byte displacement.
A 1- or 2-byte immediate operand.
A 1- or 2-byte displacement followed by a 1- or 2-byte immediate operand.
A 2-byte displacement and a 2-byte segment address (for direct intersegment addressing only).

Which of these possibilities is used is determined by the op code and addressing mode. If a displacement or immediate operand is 2 bytes long, the low-order byte always appears first.

The op code usually occupies the first byte, and only the first byte, of an instruction, but there are a few instructions in which a register designation is in the first byte and a few other instructions in which 3 bits of the op code are in the second byte. Within most of the op codes there are special 1-bit indicators. They are:

W-bit—If an instruction can operate on either a byte or a word, the op code includes a W-bit which indicates whether a byte (W = 0) or a word (W = 1) is being accessed.

D-bit—For double-operand instructions (except for instructions with one operand being immediate and string instructions, which are discussed in Chap. 5). One of the operands must be a register specified by a REG field. For these instructions the D-bit is used to indicate whether the register specified by REG is the source operand (D = 0) or the destination operand (D = 1).

S-bit—An 8-bit 2's complement number can be extended to a 16-bit 2's complement number by letting all of the bits in the high-order byte equal the MSB in the low-order byte. This is referred to as *sign extension*. The S-bit appears with the W-bit in the immediate to register/memory add, subtract, and compare instructions and is assigned as follows:

8-bit operation—S:W = 00
16-bit operation with a 16-bit immediate operand—S:W = 01
16-bit operation with a sign-extended 8-bit immediate operand—S:W = 11

For small numbers, the latter case would permit the use of a 1-byte immediate operand.

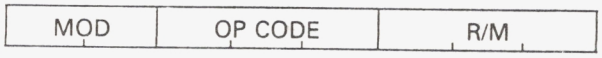
V-bit—Used by shift and rotate instructions to determine the number of shifts (see Chap. 3).

Z-bit—Used by the REP instruction (which is discussed in Chap. 5).

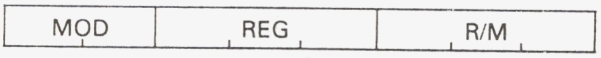
A register designation is 2 bits long if it is for a segment register and 3 bits long if it is for any other type of register. The register addresses (i.e., register bit

combination designations) are given in Fig. 2-14. There is no ambiguity in using, say, 00 to represent the segment register ES and 000 to represent both the register AX and its low-order byte AL. This is because the intended type of register is implied by the op code.

If there are two op code/addressing mode bytes, then the second byte is of one of the following two forms:



or



The first of these forms is for single-operand instructions (or instructions involving two operands with one of them being implied by the op code), and the second is for double-operand instructions, in which case REG specifies a register that is the source operand or destination operand depending on the value of the D-bit.

The operand specified by the MOD and R/M fields is determined according to the table in Fig. 2-15. If MOD $\neq$ 11, an effective address is computed according to the entries in the table. Note that MOD = 00 means that there is no displacement except when R/M = 110, in which case direct addressing is implied. MOD = 01 means that the third byte of the instruction contains an 8-bit displacement which is automatically sign-extended to 16 bits before it is used to compute the effective address, MOD = 10 means that the third and fourth bytes of the instruction contain a 16-bit displacement. For MOD = 11, the operand is in the register whose address is designated by the R/M field.

Also indicated in Fig. 2-15 is the segment register to be used for each of the MOD and R/M combinations. Although the effective address of a memory operand is determined by the MOD and R/M fields, the 20-bit physical address is found by adding the effective address to the contents of a segment register multiplied by 16. For the address modes involving the BP register, the segment register SS is added to the effective address to produce the physical address. For all other address modes, the segment register DS is used.

To permit exceptions to the segment register usage given in Fig. 2-15, a special 1-byte instruction called a *segment override prefix* is available. A segment override prefix has the form

Register Address	Registers W=1	Registers W=0	Register Address	Segment Register
000	AX	AL	00	ES
001	CX	CL	01	CS
010	DX	DL	10	SS
011	BX	BL	11	DS
100	SP	AH		
101	BP	CH		
110	SI	DH		
111	DI	BH		

Figure 2-14 Register addresses.

R/M \ MOD	00	01	10	11 W=0	11 W=1
000 Segment reg.	(BX)+(SI) DS	(BX)+(SI)+D8 DS	(BX)+(SI)+D16 DS	AL	AX
001 Segment reg.	(BX)+(DI) DS	(BX)+(DI)+D8 DS	(BX)+(DI)+D16 DS	CL	CX
010 Segment reg.	(BP)+(SI) SS	(BP)+(SI)+D8 SS	(BP)+(SI)+D16 SS	DL	DX
011 Segment reg.	(BP)+(DI) SS	(BP)+(DI)+D8 SS	(BP)+(DI)+D16 SS	BL	BX
100 Segment reg.	(SI) DS	(SI)+D8 DS	(SI)+D16 DS	AH	SP
101 Segment reg.	(DI) DS	(DI)+D8 DS	(DI)+D16 DS	CH	BP
110 Segment reg.	D16 DS	(BP)+D8 SS	(BP)+D16 SS	DH	SI
111 Segment reg.	(BX) DS	(BX)+D8 DS	(BX)+D16 DS	BH	DI

D8: 8-bit displacement (sign extended when executed)
D16: 16-bit displacement

Figure 2-15 Address modes and default segment registers for various MOD and R/M field combinations.

If an instruction is preceded by a segment override prefix, the segment register REG is used for data reference during the execution of that instruction. For the addressing modes given in the figure, DS may be overridden by CS, SS, or ES; and when BP is used, SS may be overridden by DS, CS, or ES. Specific cases that cannot involve overrides are:

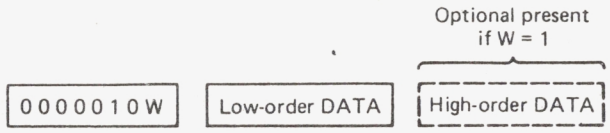
1. The CS register is always used as the segment register when computing the address of the next instruction to be executed.
2. When SP is used, SS is always the segment register. (Stack operations are discussed in Chap. 4.)
3. For string operations the ES register is always used as the segment register for the destination operand. (String operations are discussed in Chap. 5.)

To further examine the 8086's machine instructions, let us consider the addition (ADD) instruction. An ADD causes the contents of the location indicated by the source operand to be added to the contents of the location indicated by the destination operand and the sum to replace the contents of the location indicated by the destination operand. This instruction may take on one of the three forms shown in Fig. 2-16, depending on the addressing mode.

Figure 2-17 shows the machine language code for two ADD instructions, both of which add the contents of register BH to the contents of CL and put the results in CL. In the first instruction D = 1 indicates that REG = 001 = CL is where the sum will be stored. MOD = 11, so that R/M designates a register, the register

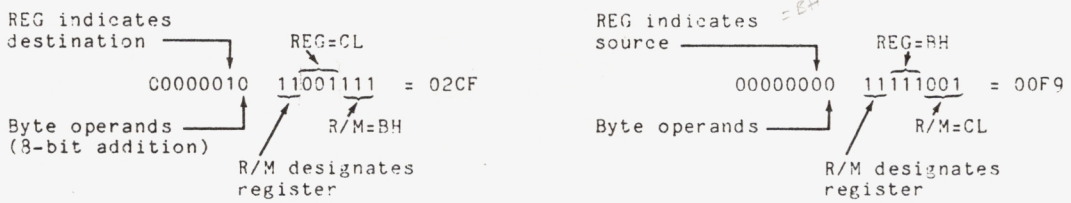
(a) Add register to register or memory and store results in register or memory

(b) Add immediate to register (memory) and put results in register (memory)



(c) Add immediate with AX(AL) and store results in AX(AL) — special case for accumulator

Figure 2-16 Formats for the ADD instruction.



(a) REG indicates destination

(b) REG indicates source

Figure 2-17 Two equivalent instructions for adding the contents of the BH register to those of the CL register.

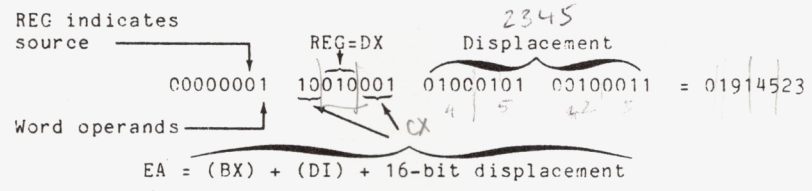
111 = BH. In the second instruction, D = 0, which causes REG = 111 = BH to be the source. Again MOD = 11 and R/M designates a register, which in this case is 001 = CL.

Figure 2-18(a) shows an instruction that uses the relative based indexed addressing mode to add a memory location with a register. From D = 0 it is seen that the sum is put in the memory location and W = 1 indicates a 16-bit addition. The effective address is found by adding the contents of BX and DI to the 16-bit displacement, which is 2345. If (BX) = 0892 and (DI) = 59A3, then

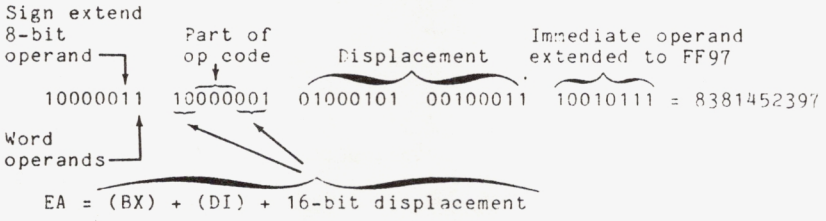
$$EA = 0892 + 59A3 + 2345 = 857A$$

Figure 2-18(a) shows an instruction that uses the relative based indexed addressing mode to add a memory location with a register. From D = 0 it is seen that the sum is put in the memory location and W = 1 indicates a 16-bit addition. The effective address is found by adding the contents of BX and DI to the 16-bit displacement, which is 2345. If (BX) = 0892 and (DI) = 59A3, then

Figure 2-18 Two examples of the ADD instruction using the relative based indexed addressing mode.



(a) Register to memory addition


Sign extend 8-bit operand
Part of op code
Displacement
Immediate operand extended to FF97
10000011 10000001 01000101 00100011 10010111 = 8381452397
Word operands
EA = (BX) + (DI) + 16-bit displacement

(b) Immediate to memory addition

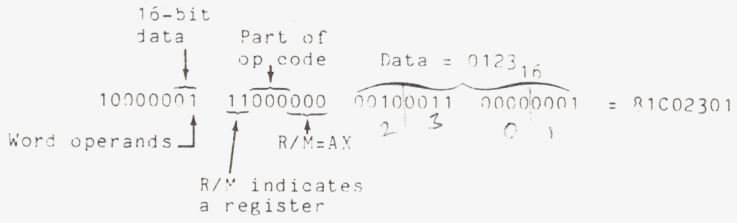


Figure 2-19 Two forms for adding an immediate operand to the AX register.

Last, several instructions have both a long form and a short form. The long and short forms of an instruction for adding an immediate operand to the AX register are given in Fig. 2-19. Because S:W = 01 there are 2 bytes of data in the instruction. In the long form AX is explicitly designated by the R/M field and in the short form AX is implied by the op code. Short forms will be discussed more thoroughly in Sec. 3-12.

Address	Contents	
00200	02	First instruction
00201	CF	
00202	01	Second instruction
00203	91	
00204	45	
00205	23	
00206	83	Third instruction
00207	81	
00208	45	
00209	23	
0020A	97	
0020B	05	Fourth instruction
0020C	23	
0020D	01	
0020E	.	Fifth instruction
.	.	
.	.	
.		

Figure 2-20 Sequence of instructions in memory.

Figure 2-20 shows how the instructions in Figs. 2-17(a), 2-18, and 2-19(b) would appear if they were placed in sequence in memory beginning at address 00200. If the contents of CS were 0018, the IP register would increment from 0080 to 008E as these instructions are executed.

2-4 INSTRUCTION EXECUTION TIMING

The execution time of an instruction can be determined by multiplying the number of clock cycles needed to execute the instruction by the clock's period. It can be expressed as the sum of a basic execution time (which varies depending on the instruction and addressing modes) plus the time required to calculate the effective address if a memory operand is involved. The basic execution time assumes that the instruction to be executed has already been prefetched and stored in the instruction queue; otherwise, any additional clock cycles necessary to fetch the instruction must be added. The basic execution times of some sample instructions for the 8086/8088 are shown in Fig. 2-21. A complete listing of execution times is

Figure 2-21 Examples of instruction execution times.

Instruction	No. of Clock Cycles	No. of Transfers
ADD (addition) or SUB (subtraction)		
Register to register	3	0
Memory to register	9+EA	1
Register to memory	16+EA	2
Immediate to register	4	0
Immediate to memory	17+EA	2
MOV (move)		
Accumulator to memory	10	1
Memory to accumulator	10	1
Register to register	2	0
Memory to register	8+EA	1
Register to memory	9+EA	1
Immediate to register	4	0
Immediate to memory	10+EA	1
Register to segment register	2	0
Memory to segment register	8+EA	1
Segment register to register	2	0
Segment register to memory	9+EA	1
MUL (unsigned multiply)		
8-bit register multiplier	70-77	0
16-bit register multiplier	118-133	0
8-bit memory multiplier	(76-83)+EA	1
16-bit memory multiplier	(124-139)+EA	1
IMUL (signed multiply)		
8-bit register multiplier	80-98	0
16-bit register multiplier	128-154	0
8-bit memory multiplier	(86-104)+EA	1
16-bit memory multiplier	(134-160)+EA	1

Instruction	No. of Clock Cycles	No. of Transfers
DIV (unsigned divide)		
8-bit register divisor	80-90	0
16-bit register divisor	144-162	0
8-bit memory divisor	(86-96)+EA	1
16-bit memory divisor	(150-168)+EA	1
IDIV (signed divide)		
8-bit register divisor	101-112	0
16-bit register divisor	165-184	0
8-bit memory divisor	(107-118)+EA	1
16-bit memory divisor	(171-190)+EA	1
Shift and rotate instructions		
Single-bit register	2	0
Variable-bit register	8+4/bit	0
Single-bit memory	15+EA	2
Variable-bit memory	20+EA+4/bit	2
JMP (unconditional branch)		
Short	15	0
Intrasegment direct	15	0
Intersegment direct	15	0
Intrasegment indirect using register mode	11	0
Intrasegment indirect	18+EA	1
Intersegment indirect	24+EA	2
Conditional branch instructions		
JCXZ	6 (no branch) 18 (branch)	0
All other conditional branch instructions	4 (no branch) 16 (branch)	0

Figure 2-21 (*continued*)

given in the Appendix. For a memory operand the effective address calculation time (also denoted EA) is determined by the addressing mode as indicated in Figure 2-22.

The third column in Fig. 2-21 indicates the number of memory references required by the instructions. To determine the execution time for an instruction in which a full word memory reference is made, we must also consider the alignment of the operand. If a word operand is located at an odd address, the 8086 must spend two bus cycles, each requiring four clock cycles, in accessing the operand. Therefore, each odd address word reference needs four extra clock cycles. For the 8088, 4 clock cycles should be added for each word transfer because the 8088 can only transfer 1 byte per bus cycle.

Note that some instructions may have several different basic execution times, depending on the addressing modes. The examples in Fig. 2-21 illustrate that for

EA	No. of Clock Cycles
Direct	6
Register indirect	5
Register relative	9
Based indexed	
(BP)+(DI) or (BX)+(SI)	7
(BP)+(SI) or (BX)+(DI)	8
Based indexed relative	
(BP)+(DI)+DISP or (BX)+(SI)+DISP	11
(BP)+(SI)+DISP or (BX)+(DI)+DISP	12

Figure 2-22 Times needed to calculate the effective address.

a given two-operand instruction, a register-to-register operation is faster than using any other addressing modes, and a register-to-memory operation is slower than a memory-to-register operation because the former requires extra time to store the result into memory. Also, some instructions have variable basic execution times which are data dependent. Typical examples are multiply, divide, shift, and rotate instructions. For conditional branch instructions, two different timing figures are given. The smaller figure is for the case in which the condition is not met, and a branch does not take place, and the larger figure corresponds to when a branch occurs. The latter case includes the flush of the instruction queue and the fetch of the next instruction.

For example, if the clock has a frequency of 5 MHz (its period is 0.2 μs), then execution times for various forms of the ADD instruction can be computed as follows:

Add register to register (result put in register) requires:

Three clock cycles for either a byte or word operand

Time = 0.6 μs

Add memory to register using based indexed relative addressing (result put in register) requires:

9 + 12 = 21 cycles for byte or word operation with word at an even address

Basic — Address calculation

Time = 4.2 μs

9 + 12 + 4 = 25 cycles if word is at an odd address

Extra cycles

Time = 5.0 μs

Add register to memory using based indexed addressing (result to memory) requires:

16 + 8 = 24 cycles for byte or word operation with word at an even address

Basic — Address calculation

Time = 4.8 μs

16 + 8 + 4 + 4 = 32 cycles if word is at an odd address

Extra cycles to retrieve word — Extra cycles to store word

Time = 6.4 μs

2-5 THE 8088

The purpose of the 8088 is to provide an 8-bit microprocessor that is fully software compatible with the 8086 (i.e., it has the same instruction set) and can be used in a hardware system that was built for an 8080 or 8085. Like the 8080 and 8085 it has 8 data lines, but its CPU architecture is essentially the same as that of the 8086. Its pin assignments are the same as the 8086 except that the 8088 address pins A15 through A8 are used only for addresses and one of the control pins, the high-byte enable pin, has been changed to a status pin because the 8088 can access only one byte at a time. Since several of the 8088's control pins are different from those of either the 8080 or the 8085, significant changes may need to be made in the bus control logic when putting an 8088 in an 8080- or 8085-based system. One must also take into account the fact that the 8088, like the 8085, differs from the 8080 in that addresses and data are multiplexed. Few or no changes should be required in the memory or I/O subsystem unless the system is to take advantage of the 8088's expanded capacity due to its 20 address lines.

The only internal difference between the 8088 and the 8086 is that the 8088 has only a 4-byte instruction queue instead of a 6-byte queue. The reason for the smaller queue is that the 8088 can only fetch one byte at a time and the longer fetch times mean that the processor cannot fully utilize a 6-byte queue. The pre-fetching algorithm is different in that instead of waiting for a 2-byte space, the 8088 will fetch an instruction byte when an empty byte in the queue becomes available. The 8088 does retain the segment registers, 20-bit addressing, and the features of the 8086 that support multiprocessing. The ability to manipulate 16-bit operands is also preserved.

BIBLIOGRAPHY

1. Morse, Stephen P., Bruce W. Ravenel, Stanley Mazor, and William P. Pohlman, "Intel Microprocessors—8008 to 8086," *Computer*, no. 10 (October, 1980), 42–60.
2. *MCS-86 User's Manual* (Santa Clara, Calif.: Intel Corporation, 1979).

EXERCISES

1. If a physical branch address is 5A230 when (CS) = 5200, what will it be if the (CS) are changed to 7800?

2. Given that the EA of a datum is 2359 and the (DS) = 490B, what is the physical address of the datum?

3. Given that

(BX) = 637D (SI) = 2A9B Displacement = C237

determine the effective address (if applicable) resulting from these registers and the addressing mode:

(a) Immediate. **(b)** Direct.
(c) Register using BX. **(d)** Register indirect using BX.
(e) Register relative using BX. **(f)** Based indexed.
(g) Based indexed relative.

4. Given that

(IP) = 2BC0 (CS) = 0200 Displacement = 5119
(BX) = 1200 (DS) = 212A (224A0) = 0600 (275B9) = 098A

find the branch address for a branch instruction that uses:

(a) Intrasegment direct addressing.
(b) Intrasegment indirect addressing which uses the BX register and register addressing.
(c) Intrasegment indirect addressing which uses the BX register and register relative addressing.

5. Give the sum and the flag settings for AF, SF, ZF, CF, OF, and PF after hexadecimally adding 62A0 to each of the following:

(a) 1234 **(b)** 4321 **(c)** CFA0 **(d)** 9D60

6. Give the difference and the flag settings for SF, ZF, CF, OF, and PF after subtracting 4AE0 from each of the following:

(a) 1234 **(b)** 5D90 **(c)** 9090 **(d)** EA04

7. Use the examples in Sec. 2-3-2 to find the machine language code for the instruction that:

(a) Adds the contents of BX to the contents of DX and puts the result in DX.
(b) Uses registers BX and SI and based indexed addressing to add a byte in memory to the (AL) and put the result in AL.
(c) Uses register BX and a displacement of B2 and register relative addressing to add

a memory location to the (CX), and then puts the result back into the memory location.

(d) Uses a displacement of 0524 and direct addressing to add the number 2A59 to a memory location and puts the sum back into the memory location.

(e) Adds the number B5 to the (AL) and puts the sum in AL. Give both a long form and a short form for this instruction.

8. Assume a 5-MHz clock and use Fig. 2-21 to determine the execution times of an instruction which:

(a) Adds the (BX) to the (CX).

(b) Uses register relative addressing to subtract the contents of a word in memory from the (AX) and puts the result in AX. First assume that the word has an even address and then assume that it has an odd address.

(c) Uses direct addressing to move a byte from memory to DX.

(d) Performs a conditional branch. First assume that the branch is taken and then assume that it is not taken.

allow significantly more complicated programs to be written, and Secs. 3-6 through 3-9 introduce the no operation and halt, flag manipulation, logical, and shift and rotate instructions. Next, directives and the overall structure of a program are considered. Section 3-11 discusses the process of assembling a program and the last section considers the translation of individual assembler language instructions into their machine language formats.

3-1 ASSEMBLER INSTRUCTION FORMAT

The general format of an assembler instruction is

```
Label:    Mnemonic    Operand,Operand    ;Comments
```

where the inclusion of spaces is arbitrary, except that at least one space must be inserted if no space would lead to an ambiguity (e.g., between the mnemonic and first operand). Also, there can be no spaces within a mnemonic or identifier and spaces within string constants or comments will be included as space characters. A *label* is an identifier that is assigned the address of the first byte of the instruction in which it appears. The presence of a label in an instruction is optional, but, if present, the label provides a symbolic name that can be used in branch instructions to branch to the instruction. If there is no label, then the colon must be deleted. All instructions must contain a mnemonic. (Mnemonics may be modified by prefixes—these prefixes are introduced as the need arises.) The presence of the operands depends on the instruction. Some instructions have no operands, some have one, and some have two. If there are two operands, they are separated by a comma. The comments field is for commenting the program and may contain any combination of characters. It is optional and if it is deleted the semicolon may also be deleted. A comment may appear on a line by itself provided that the first character on the line is a semicolon. An instruction may be continued to other lines by placing an ampersand at the beginning of each of the continuation lines.

An assembler language instruction must have an operand for each machine language instruction operand and the notation for each operand must be sufficient to indicate the operand's addressing mode. If there are two operands, the destination operand appears first and the source operand second. A typical assembler instruction, one which uses the register relative and register addressing modes to add a memory location to the register AX, is given in Fig. 3-2. When an operand is a word in memory the low-order byte of the word will have the lower address and the high-order byte the higher address, e.g., if the identifier COST is used as a word operand, then COST is associated with the low-order byte and COST + 1 with the high-order byte.

The syntax for the various types of operands is summarized in Fig. 3-3. These forms may be extended by modifiers that are introduced later, but these basic forms are all that are needed at this point. A *variable* is an identifier that is associated with the first byte of a datum. An expression is a concatenation of symbols known

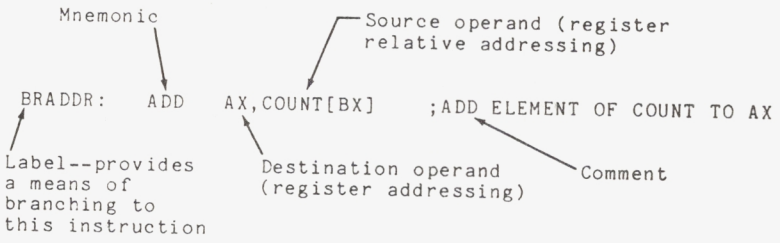

Mnemonic
Source operand (register relative addressing)
BRADDR: ADD AX,COUNT[BX] ;ADD ELEMENT OF COUNT TO AX
Label--provides a means of branching to this instruction
Destination operand (register addressing)
Comment

Figure 3-2 Representative assembler language instruction.

as *tokens*, and a constant expression is an expression which can be evaluated to produce a number. The tokens may be variable identifiers or:

Constant—A number whose base is indicated by a suffix as follows:

B—binary
D—decimal
O—octal
H—hexadecimal

The default is decimal. The first digit in a hexadecimal number must be 0 through 9; therefore, if the most significant digit is a letter (A–F), then it must be prefixed with a 0. Examples are:

$$1011_2 = 1011B$$

$$223_{10} = 223D = 223$$

$$B25A_{16} = 0B25AH$$

String Constant—A character string enclosed in single quotes (').

Arithmetic Operators—The operators "+", "−", "*", and "/".

Logical Operators—The operators "AND", "OR", "NOT", and "XOR" (exclusive OR). The logical operations are performed by putting the operands in binary form and performing the operation on the corresponding pairs of bits.

Subexpressions—An expression that is part of another expression and is delimited from its parent expression by parentheses.

Name—An identifier that represents a constant, string constant, or expression.

(The ASM-86 assembler also permits relational operations, but they are not used in this book.) The order of precedence for operators is the same as it is in high-level languages. An identifier, be it a label, variable, or name, consists of a string of up to 31 characters (actually, it may be longer but the ASM-86 assembler recognizes only the first 31 characters). The characters must be letters, digits, question marks, underscores, or "@" signs except that the first character cannot

Addressing Mode	Format	Examples
Data-related		
Immediate	Constant Expression	10110B 529 0A9H 'AB' YY-XX+5
Direct	Variable ± Constant Expression	WGT CNT-5 ARRY+5
Register	Register	AX DH
Register indirect	[Register]	[BX]
Register relative	Variable[Register ± Constant Exp] or [Register ± Constant Exp.]	VAR_X[BX] MESG[SI+10H] [BX-1]
Based indexed	[Base register][Index register]	[BP][DI]
Relative based indexed	Var.[Base reg. ± Const. Exp][Index reg. ± Const. Exp] or [Base reg. ± Const. Exp][Index reg. ± Const. Exp]	E[BX+5][SI-2] DATA[BX][SI] [BP+2][DI-9]
Branch-related		
Intrasegment direct	Label ± Constant Expression	OVFL_ERROR+7
Intrasegment indirect	Same as data-related formats except cannot be immediate	
Intersegment direct	Label ± Constant Expression	BRANCH_EXIT
Intersegment indirect	Same as data related formats except must not be immediate or register	

Note: For all but the immediate mode the "Constant Expression" may or may not be present.

Figure 3-3 Operand formats for the addressing modes.

be a digit. Identifiers cannot be keywords used by the assembler, such as instruction mnemonics, register names, and special operators. Several complete ADD instructions, along with the machine language instructions that they are translated into, are given in hexadecimal in Fig. 3-4.

It should be emphasized that a segment register designation can appear in an operand of only certain instructions. *A segment register can be explicitly specified in only the* MOV, PUSH, and POP *instructions*. The first of these instructions is discussed in the next section and the other two are defined in Chap. 4. Other instructions, such as the branch instructions, may implicitly reference the segment registers, however. Except for string operations, for a double-operand instruction one of the operands must be a register unless the source operand is immediate. In no case can IP be specified as an operand because only a branch or other control transfer instruction can change the contents of this register.

There are several modifiers, referred to as *attribute operators,* that will be

```
Machine Code                    Assembler Code

03 C3                ADD    AX,BX          ;(BX) ARE ADDED TO AX
81 C3 6D 02          ADD    BX,621         ;ADD 621 TO BX
01 11                ADD    [BX][DI],DX    ;ADD (DX) TO THE LOCATION
                                           ;WHOSE OFFSET IS THE SUM OF
                                           ;(BX) AND (DI)
02 86 2E 02          ADD    AL,NUM[BP+11];ADD TO AL THE CONTENTS OF THE
                                           ;LOCATION WHOSE OFFSET IS
                                           ;THE SUM OF THE OFFSET OF
                                           ;NUM,(BP), AND 11

Machine code assumes the offset of NUM is 0223 so that
      EA = (BP) + 0223 + 000B = (BP) + 022E
```

Figure 3-4 Typical assembler language instructions.

discussed as the book progresses. One of these operators, the pointer (PTR) operator, needs to be introduced at this point. Many instructions can operate on either bytes or words, depending on whether or not the W-bit is set. Normally, at least one of the operands is such that a byte or a word operation is implied by its notation or by how a memory operand is defined, and the assembler sets or clears the W-bit accordingly, e.g., because of the destination operand the assembler will assume that

```
ADD    AX,[BX]
```

is a word addition and

```
ADD    AL,[BX]
```

a byte addition. Also, if appropriate directives are used to define COST to be a word variable and COUNT to be a byte variable, then

```
INC    COST
```

will be a word operation, and

```
INC    COUNT
```

will be a byte operation. For some situations, however, it is impossible for the assembler to deduce the operand type. The instruction

```
INC    [BX]
```

increments the quantity whose address is in BX, but should it increment a byte or a word? One of the purposes of the PTR operator is to specify the length of a quantity in this and other ambiguous situations. It is applied by writing the desired type followed by PTR. For the above INC instruction the PTR operator would be used to modify the operand as follows:

```
INC    BYTE PTR    [BX]
```

if a byte is to be incremented, or

```
INC    WORD PTR [BX]
```

if a word is to be incremented.

3-2 DATA TRANSFER INSTRUCTIONS

All computers need a variety of instructions solely for moving data, addresses, and immediate operands into register or memory locations. It is the purpose of this section to introduce these instructions for the 8086, but before this is done let us consider the notation that is used throughout this book to define instructions. The symbols and abbreviations to be employed are summarized in Fig. 3-5 (many of them have already been introduced) and are to be used to provide shortened instruction definitions and other descriptive statements. Using this notation the statement

> The contents of register AX are replaced by the contents of register AX plus the immediate operand.

Figure 3-5 Glossary of symbols and abbreviations.

Abbreviation	Meaning	Symbol	Meaning
OPR	Operand	←	Replaced by
SRC	Source operand	↔	Interchanged with
DST	Destination operand	(X)	Contents of X
REG	Register	$\wedge$	Logical AND
RSRC	Source register	$\vee$	Logical OR
RDST	Destination register	$\veebar$	Logical exclusive OR
CNT	Count	$\overline{X}$	Complement of X
DISP	Displacement (general)		
D8	8-bit displacement		
D16	16-bit displacement		
ADDR	Address		
EA	Effective address		
SEG	Segment		
DATA	Immediate operand (general)		
DATA8	8-bit immediate operand		
DATA16	16-bit immediate operand		
AX,BX,CX, DX,AL,AH, BL,BH,CL, CH,DL,DH, IP,BP,SP, SI,DI	8086 register names		
CS,SS,DS ES	8086 segment register names		
DF,IF,TF	8086 control flag names		
OF,SF,ZF, AF,PF,CF	8086 condition flag names		

could be written

$$(AX) \leftarrow (AX) + DATA$$

Also, the statement

> The contents of BX are replaced by the contents of the memory location whose effective address is the contents of BP plus a displacement.

could be written

$$(BX) \leftarrow ((BP) + DISP)$$

There are four basic 8086 instructions for transferring quantities to and/or from the registers and memory; they are the MOV, LEA, LDS, and LES instructions and are defined in Fig. 3-6. The MOV instruction is for moving a byte or word within the CPU or between the CPU and memory. Depending on the addressing modes it can transfer information from a:

- Register to a register.
- Immediate operand to a register.
- Immediate operand to a memory location.
- Memory location to a register.
- Register to a memory location.
- Register/memory location to a segment register (except CS).
- A segment register to a register/memory location.

None of the flags are changed by the execution of a move instruction. Several examples of MOV instructions are given in Fig. 3-7.

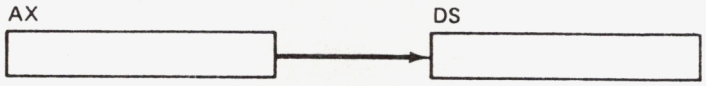
Figure 3-6 Data transfer instructions.

Name	Mnemonic and Format		Description
Move	MOV	DST,SRC	(DST)←(SRC)
Load effective address	LEA	REG,SRC	(REG)←SRC
Load DS with pointer	LDS	REG,SRC	(REG)←(SRC) (DS)←(SRC+2)
Load ES with pointer	LES	REG,SRC	(REG)←(SRC) (ES)←(SRC+2)
Exchange	XCHG	OPR1,OPR2	(OPR1)←→(OPR2)

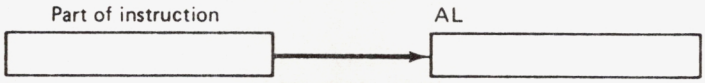
Flags: None of the flags are affected.

Addressing modes: The destination cannot be immediate and cannot be CS. For the LEA, LDS, and LES instructions REG cannot be a segment register and the source mode cannot be the immediate or register modes. For MOV, unless the source operand is immediate, one of the operands must be a register. For XCHG, at least one of the operands must be a register, but neither operand can be a segment register.

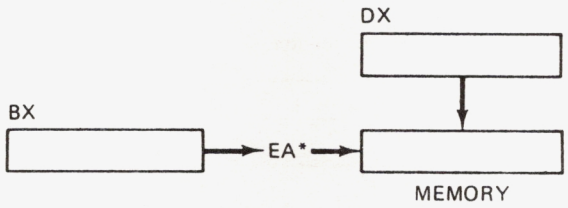
```
MOV   DS, AX          ; MOVE (AX) TO DS
```



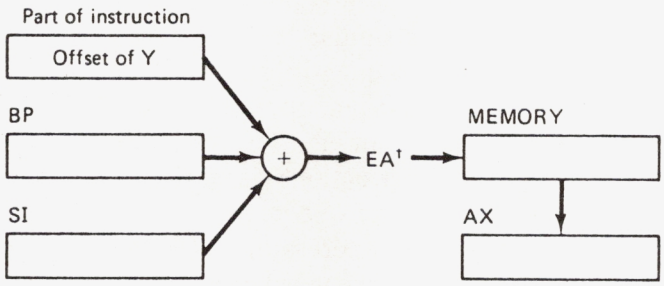

```
MOV   AL, 'E'         ; MOVE ASCII 'E' TO AL
```




```
MOV   [BX], DX        ; MOVE (DX) INTO LOCATION INDICATED
                      ; BY (BX)
```




```
MOV   AX, Y[BP][SI]   ; MOVE INTO AX THE CONTENTS OF THE
                      ; LOCATION FOUND BY ADDING (BP), (SI), AND
                      ; THE OFFSET OF Y
```



*(DS) X 16_{10} is added to the EA.
†(SS) X 16_{10} is added to the EA.

Figure 3-7 Examples of the MOV instruction.

A program sequence for interchanging the contents of the two bytes at the memory locations represented by OPR1 and OPR2 [i.e., (OPR1) ↔ (OPR2)] is shown in Fig. 3-8(a) and the action taken by the sequence is shown in Fig. 3-8(b). Figure 3-9 shows how the machine code for this segment of assembler code is translated and stored in memory. The indicated instruction addresses assume that the first instruction begins at 0230 and (CS) = 0C30, and the offsets of OPR1 and OPR2 are assumed to be 1200 and 1C20.

A typical LEA instruction is

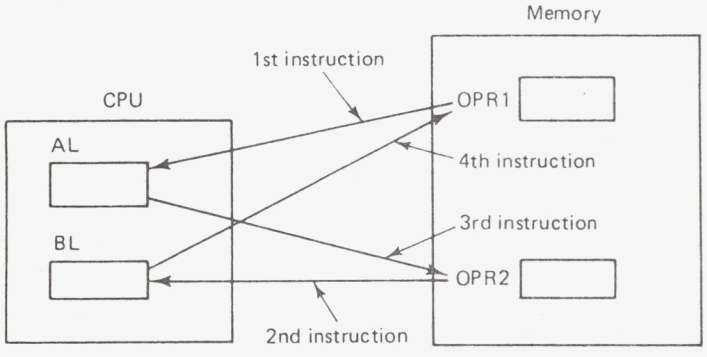
```
LEA     SI,COL[BX]
```

```
Data-related directives, input, and other code
OPR1 and OPR2 are defined as byte variables

        MOV     AL,OPR1         ;(OPR1) TO AL
        MOV     BL,OPR2         ;(OPR2) TO BL
        MOV     OPR2,AL         ;(AL) TO OPR2
        MOV     OPR1,BL         ;(BL) TO OPR1

Output and other code
```

(a) Code



(b) Action taken

Figure 3-8 Program sequence for interchanging the contents of two locations.

This instruction loads the effective address computed by adding the contents of BX to the offset of COL into SI. The LEA instruction does not affect any of the flags. Its source operand must produce an effective address and it is this address, *not* the contents of this address, that is put into the destination register.

The XCHG instruction is similar to MOV, but instead of duplicating the source in the destination the two quantities are interchanged. Because immediate operands cannot be changed, neither operand in the XCHG instruction can be immediate. In the example shown in Fig. 3-8, the XCHG instruction could be used to replace two of the MOV instructions—see Fig. 3-10. Note that by applying XCHG only three instructions and one register are required. The following sequence of instructions causes the address of ARRAY to be put into BX, 0 to be loaded into SI, and the contents of the word beginning at ARRAY to be transferred to AX:

```
LEA     BX,ARRAY
MOV     SI,0
MOV     AX,[BX][SI]
```

The LDS and LES instructions are the same except that the former loads the DS register from memory and the latter loads ES from memory. Both instructions also load a second nonsegment register from memory and neither instruction affects the flags. Typical LDS and LES instructions are:

```
LDS     SI,STRING_SOURCE_POINTER
LES     DI,TABLE[BX]
```

where STRING_SOURCE_POINTER and TABLE are double-word variables. These instructions facilitate switching data segments by simultaneously loading a

Physical Address	Machine Code	
0C530	8A	MOV AL,OPR1
0C531	06	
0C532	00	
0C533	12	
0C534	8A	MOV BL,OPR2
0C535	1E	
0C536	20	
0C537	1C	
0C538	88	MOV OPR2,AL
0C539	06	
0C53A	20	
0C53B	1C	
0C53C	88	MOV OPR1,BL
0C53D	1E	
0C53E	00	
0C53F	12	
0C540	.	
.	.	
.	.	
.		

Figure 3-9 Machine language code for the program sequence given in Fig. 3-8(a).

base or index register and a segment register. The usefulness of their applications will become apparent later.

3-3 ARITHMETIC INSTRUCTIONS

The 8086 instruction set includes instructions for performing arithmetic operations in binary, packed BCD, and unpacked BCD. Figure 3-11 outlines the arithmetic operations that are implemented by 8086 instructions. As indicated in this figure,

Figure 3-10 Interchanging bytes in memory using XCHG.

```
Data-related directives, input, and other code
OPR1 and OPR2 are defined as byte variables

        MOV     AL,OPR1         ;LOAD AL WITH (OPR1)
        XCHG    AL,OPR2         ;EXCHANGE (AL) WITH (OPR2)
        MOV     OPR1,AL         ;MOVE (AL) TO (OPR1)

Output and other code
```

The cross-reference symbol table summarizes the information regarding the identifiers in the program. For each segment name this table indicates the size of the segment in hexadecimal and the line numbers of the lines in which the name appears. The line number of the segment directive containing the name is marked with the symbol "#". The variables and labels are accompanied by letters indicating whether they are a variable (V) or label (L), an attribute designation, the offset assigned to them, the name of the segment in which they are defined, and the line numbers of the lines in which they appear. As with the segment names, the lines in which they are defined are marked with "#" signs. For example, OPER2 is a word variable that is defined in the segment DATA_SEG. Its offset within DATA_SEG is 0002H. It appears in lines 3 and 11 of the listing and is defined in line 3. Similarly, STORE is seen to be a near label that is defined in CODE_SEG and has an offset of 0010H within CODE_SEG. It appears in lines 12 and 14 and is defined in line 14. The reader should carefully verify all entries in the symbol table and the machine code in the program listing.

3-12 TRANSLATION OF ASSEMBLER INSTRUCTIONS

The translation from assembler instructions to machine instructions is, in most cases, quite straightforward. All of the 8086 machine instructions consist of 1 or 2 bytes of op code and addressing mode designations with from 0 to 4 bytes of immediate, displacement, or segment address information appended to them. The number of appended bytes depends on the addressing modes. If the addressing modes call for both an immediate operand and a displacement the displacement will appear first, and if both an offset and a segment address are present, the offset will appear first.

The machine code formats of all of the 8086/8088 instructions, including those that are defined in the later chapters, are given in Fig. 3-71. For convenience, a summary of the register addresses, addressing modes, etc., is also included in this figure. All but one of the double-operand instructions (MOV, ADD, ADC, SUB, SBB, CMP, AND, OR, XOR, and TEST) in which the source operand is immediate and the other operand is the AX or AL register (i.e., accumulator) have both a long form and a short form. In the short form the AX (for word operations) or AL (for byte operations) is implied by the op code. In addition, the MOV instruction has a short form for any immediate to register transfer. The XCHG instruction has a short form if the exchange is between a register and AX or AL. Several of the single-operand instructions (INC, DEC, PUSH, and POP) have short forms if the operand is a register.

In all cases the short form is one byte shorter than the corresponding long form, which in some instances implies that the instruction is only 1 byte in length. As a general rule, the assembler will always assemble the short form, if possible. For example,

```
CMP     AX,5A2H
```

Figure 3-71 Machine code for the 8086/8088 instructions. (Reprinted by permission of Intel Corporation. Copyright 1979.)

AL = 8-bit accumulator
AX = 16-bit accumulator
CX = Count register
DS = Data segment
ES = Extra segment
Above/below refers to unsigned value
Greater = more positive;
Less = less positive (more negative) signed values
if d = 1 then "to" reg; if d = 0 then "from" reg
if w = 1 then word instruction; if w = 0 then byte instruction

if s:w = 01 then 16 bits of immediate data form the operand
if s:w = 11 then an immediate data byte is sign extended to form the 16-bit operand
if v = 0 then "count" = 1; if v = 1 then "count" in (CL)
x = don't care
z is used for string primitives for comparison with Z-FLAG

SEGMENT OVERRIDE PREFIX

0 0 1 reg 1 1 0

if mod = 11 then r/m is treated as a REG field
if mod = 00 then DISP = 0*, disp-low and disp-high are absent
if mod = 01 then DISP = disp-low sign-extended to 16-bits, disp-high is absent
if mod = 10 then DISP = disp-high: disp-low
if r/m = 000 then EA = (BX) + (SI) + DISP
if r/m = 001 then EA = (BX) + (DI) + DISP
if r/m = 010 then EA = (BP) + (SI) + DISP
if r/m = 011 then EA = (BP) + (DI) + DISP
if r/m = 100 then EA = (SI) + DISP
if r/m = 101 then EA = (DI) + DISP
if r/m = 110 then EA = (BP) + DISP*
if r/m = 111 then EA = (BX) + DISP
DISP follows 2nd byte of instruction (before data if required)

*except if mod = 00 and r/m = 110 then EA = disp-high: disp-low

REG is assigned according to the following table

16-Bit (w = 1)	8-Bit (w = 0)	Segment
000 AX	000 AL	00 ES
001 CX	001 CL	01 CS
010 DX	010 DL	10 SS
011 BX	011 BL	11 DS
100 SP	100 AH	
101 BP	101 CH	
110 SI	110 DH	
111 DI	111 BH	

Instructions which reference the flag register file as a 16-bit object use the symbol FLAGS to represent the file:

FLAGS = X:X:X:X:(OF):(DF):(IF):(TF):(SF):(ZF):X:(AF):X:(PF):X:(CF)

8086/8088 Instruction Encoding

DATA TRANSFER

MOV = Move:	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Register/memory to/from register	1 0 0 0 1 0 d w	mod reg r/m	(DISP-LO)	(DISP-HI)		
Immediate to register/memory	1 1 0 0 0 1 1 w	mod 0 0 0 r/m	(DISP-LO)	(DISP-HI)	data	data if w = 1
Immediate to register	1 0 1 1 w reg	data	data if w = 1			
Memory to accumulator	1 0 1 0 0 0 0 w	addr-lo	addr-hi			
Accumulator to memory	1 0 1 0 0 0 1 w	addr-lo	addr-hi			
Register/memory to segment register	1 0 0 0 1 1 1 0	mod 0 SR r/m	(DISP-LO)	(DISP-HI)		
Segment register to register/memory	1 0 0 0 1 1 0 0	mod 0 SR r/m	(DISP-LO)	(DISP-HI)		

PUSH = Push:

Register/memory	1 1 1 1 1 1 1 1	mod 1 1 0 r/m	(DISP-LO)	(DISP-HI)
Register	0 1 0 1 0 reg			
Segment register	0 0 0 reg 1 1 0			

POP = Pop:

Register/memory	1 0 0 0 1 1 1 1	mod 0 0 0 r/m	(DISP-LO)	(DISP-HI)
Register	0 1 0 1 1 reg			
Segment register	0 0 0 reg 1 1 1			

Figure 3-71 Continued.

DATA TRANSFER (Cont'd.)

	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
XCHG = Exchange:						
Register/memory with register	1 0 0 0 0 1 1 w	mod reg r/m	(DISP-LO)	(DISP-HI)		
Register with accumulator	1 0 0 1 0 reg					
IN = Input from:						
Fixed port	1 1 1 0 0 1 0 w	DATA-8				
Variable port	1 1 1 0 1 1 0 w					
OUT = Output to:						
Fixed port	1 1 1 0 0 1 1 w	DATA-8				
Variable port	1 1 1 0 1 1 1 w					
XLAT = Translate byte to AL	1 1 0 1 0 1 1 1					
LEA = Load EA to register	1 0 0 0 1 1 0 1	mod reg r/m	(DISP-LO)	(DISP-HI)		
LDS = Load pointer to DS	1 1 0 0 0 1 0 1	mod reg r/m	(DISP-LO)	(DISP-HI)		
LES = Load pointer to ES	1 1 0 0 0 1 0 0	mod reg r/m	(DISP-LO)	(DISP-HI)		
LAHF = Load AH with flags	1 0 0 1 1 1 1 1					
SAHF = Store AH into flags	1 0 0 1 1 1 1 0					
PUSHF = Push flags	1 0 0 1 1 1 0 0					
POPF = Pop flags	1 0 0 1 1 1 0 1					

ARITHMETIC

ADD = Add:						
Reg/memory with register to either	0 0 0 0 0 0 d w	mod reg r/m	(DISP-LO)	(DISP-HI)		
Immediate to register/memory	1 0 0 0 0 0 s w	mod 0 0 0 r/m	(DISP-LO)	(DISP-HI)	data	data if s: w=01
Immediate to accumulator	0 0 0 0 0 1 0 w	data	data if w=1			
ADC = Add with carry:						
Reg/memory with register to either	0 0 0 1 0 0 d w	mod reg r/m	(DISP-LO)	(DISP-HI)		
Immediate to register/memory	1 0 0 0 0 0 s w	mod 0 1 0 r/m	(DISP-LO)	(DISP-HI)	data	data if s: w=01
Immediate to accumulator	0 0 0 1 0 1 0 w	data	data if w=1			
INC = Increment:						
Register/memory	1 1 1 1 1 1 1 w	mod 0 0 0 r/m	(DISP-LO)	(DISP-HI)		
Register	0 1 0 0 0 reg					
AAA = ASCII adjust for add	0 0 1 1 0 1 1 1					
DAA = Decimal adjust for add	0 0 1 0 0 1 1 1					

Figure 3-71 Continued.

ARITHMETIC (Cont'd.)

	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
SUB = Subtract:						
Reg/memory and register to either	0 0 1 0 1 0 d w	mod reg r/m	(DISP-LO)	(DISP-HI)		
Immediate from register/memory	1 0 0 0 0 0 s w	mod 1 0 1 r/m	(DISP-LO)	(DISP-HI)	data	data if s: w=01
Immediate from accumulator	0 0 1 0 1 1 0 w	data	data if w=1			
SBB = Subtract with borrow:						
Reg/memory and register to either	0 0 0 1 1 0 d w	mod reg r/m	(DISP-LO)	(DISP-HI)		
Immediate from register/memory	1 0 0 0 0 0 s w	mod 0 1 1 r/m	(DISP-LO)	(DISP-HI)	data	data if s: w=01
Immediate from accumulator	0 0 0 1 1 1 0 w	data	data if w=1			
DEC Decrement:						
Register/memory	1 1 1 1 1 1 1 w	mod 0 0 1 r/m	(DISP-LO)	(DISP-HI)		
Register	0 1 0 0 1 reg					
NEG Change sign	1 1 1 1 0 1 1 w	mod 0 1 1 r/m	(DISP-LO)	(DISP-HI)		
CMP = Compare:						
Register/memory and register	0 0 1 1 1 0 d w	mod reg r/m	(DISP-LO)	(DISP-HI)		
Immediate with register/memory	1 0 0 0 0 0 s w	mod 1 1 1 r/m	(DISP-LO)	(DISP-HI)	data	data if s: w=1
Immediate with accumulator	0 0 1 1 1 1 0 w	data				
AAS ASCII adjust for subtract	0 0 1 1 1 1 1 1					
DAS Decimal adjust for subtract	0 0 1 0 1 1 1 1					
MUL Multiply (unsigned)	1 1 1 1 0 1 1 w	mod 1 0 0 r/m	(DISP-LO)	(DISP-HI)		
IMUL Integer multiply (signed)	1 1 1 1 0 1 1 w	mod 1 0 1 r/m	(DISP-LO)	(DISP-HI)		
AAM ASCII adjust for multiply	1 1 0 1 0 1 0 0	0 0 0 0 1 0 1 0	(DISP-LO)	(DISP-HI)		
DIV Divide (unsigned)	1 1 1 1 0 1 1 w	mod 1 1 0 r/m	(DISP-LO)	(DISP-HI)		
IDIV Integer divide (signed)	1 1 1 1 0 1 1 w	mod 1 1 1 r/m	(DISP-LO)	(DISP-HI)		
AAD ASCII adjust for divide	1 1 0 1 0 1 0 1	0 0 0 0 1 0 1 0	(DISP-LO)	(DISP-HI)		
CBW Convert byte to word	1 0 0 1 1 0 0 0					
CWD Convert word to double word	1 0 0 1 1 0 0 1					
LOGIC						
NOT Invert	1 1 1 1 0 1 1 w	mod 0 1 0 r/m	(DISP-LO)	(DISP-HI)		
SHL/SAL Shift logical/arithmetic left	1 1 0 1 0 0 v w	mod 1 0 0 r/m	(DISP-LO)	(DISP-HI)		
SHR Shift logical right	1 1 0 1 0 0 v w	mod 1 0 1 r/m	(DISP-LO)	(DISP-HI)		
SAR Shift arithmetic right	1 1 0 1 0 0 v w	mod 1 1 1 r/m	(DISP-LO)	(DISP-HI)		
ROL Rotate left	1 1 0 1 0 0 v w	mod 0 0 0 r/m	(DISP-LO)	(DISP-HI)		

Figure 3-71 Continued.

LOGIC (Cont'd.)	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
ROR Rotate right	1 1 0 1 0 0 v w	mod 0 0 1 r/m	(DISP-LO)	(DISP-HI)		
RCL Rotate through carry flag left	1 1 0 1 0 0 v w	mod 0 1 0 r/m	(DISP-LO)	(DISP-HI)		
RCR Rotate through carry right	1 1 0 1 0 0 v w	mod 0 1 1 r/m	(DISP-LO)	(DISP-HI)		

AND = And:						
Reg/memory with register to either	0 0 1 0 0 0 d w	mod reg r/m	(DISP-LO)	(DISP-HI)		
Immediate to register/memory	1 0 0 0 0 0 0 w	mod 1 0 0 r/m	(DISP-LO)	(DISP-HI)	data	data if w=1
Immediate to accumulator	0 0 1 0 0 1 0 w	data	data if w=1			

TEST = And function to flags no result:						
Register/memory and register	0 0 0 1 0 0 d w	mod reg r/m	(DISP-LO)	(DISP-HI)		
Immediate data and register/memory	1 1 1 1 0 1 1 w	mod 0 0 0 r/m	(DISP-LO)	(DISP-HI)	data	data if w=1
Immediate data and accumulator	1 0 1 0 1 0 0 w	data				

OR = Or:						
Reg/memory and register to either	0 0 0 0 1 0 d w	mod reg r/m	(DISP-LO)	(DISP-HI)		
Immediate to register/memory	1 0 0 0 0 0 0 w	mod 0 0 1 r/m	(DISP-LO)	(DISP-HI)	data	data if w=1
Immediate to accumulator	0 0 0 0 1 1 0 w	data	data if w=1			

XOR = Exclusive or:						
Reg/memory and register to either	0 0 1 1 0 0 d w	mod reg r/m	(DISP-LO)	(DISP-HI)		
Immediate to register/memory	0 0 1 1 0 1 0 w	data	(DISP-LO)	(DISP-HI)	data	data if w=1
Immediate to accumulator	0 0 1 1 0 1 0 w	data	data if w=1			

STRING MANIPULATION	
REP = Repeat	1 1 1 1 0 0 1 z
MOVS = Move byte/word	1 0 1 0 0 1 0 w
CMPS = Compare byte/word	1 0 1 0 0 1 1 w
SCAS = Scan byte/word	1 0 1 0 1 1 1 w
LODS = Load byte/wd to AL/AX	1 0 1 0 1 1 0 w
STDS = Stor byte/wd from AL/A	1 0 1 0 1 0 1 w

Figure 3-71 Continued.

CONTROL TRANSFER

CALL = Call:	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Direct within segment	1 1 1 0 1 0 0 0	IP-INC-LO	IP-INC-HI			
Indirect within segment	1 1 1 1 1 1 1 1	mod 0 1 0 r/m	(DISP-LO)	(DISP-HI)		
Direct intersegment	1 0 0 1 1 0 1 0	IP-lo	IP-hi			
		CS-lo	CS-hi			
Indirect intersegment	1 1 1 1 1 1 1 1	mod 0 1 1 r/m	(DISP-LO)	(DISP-HI)		

JMP = Unconditional Jump:				
Direct within segment	1 1 1 0 1 0 0 1	IP-INC-LO	IP-INC-HI	
Direct within segment-short	1 1 1 0 1 0 1 1	IP-INC8		
Indirect within segment	1 1 1 1 1 1 1 1	mod 1 0 0 r/m	(DISP-LO)	(DISP-HI)
Direct intersegment	1 1 1 0 1 0 1 0	IP-lo	IP-hi	
		CS-lo	CS-hi	
Indirect intersegment	1 1 1 1 1 1 1 1	mod 1 0 1 r/m	(DISP-LO)	(DISP-HI)

RET = Return from CALL:			
Within segment	1 1 0 0 0 0 1 1		
Within seg adding immed to SP	1 1 0 0 0 0 1 0	data-lo	data-hi
Intersegment	1 1 0 0 1 0 1 1		
Intersegment adding immediate to SP	1 1 0 0 1 0 1 0	data-lo	data-hi
JE/JZ = Jump on equal/zero	0 1 1 1 0 1 0 0	IP-INC8	
JL/JNGE = Jump on less/not greater or equal	0 1 1 1 1 1 0 0	IP-INC8	
JLE/JNG = Jump on less or equal/not greater	0 1 1 1 1 1 1 0	IP-INC8	
JB/JNAE = Jump on below/not above or equal	0 1 1 1 0 0 1 0	IP-INC8	
JBE/JNA = Jump on below or equal/not above	0 1 1 1 0 1 1 0	IP-INC8	
JP/JPE = Jump on parity/parity even	0 1 1 1 1 0 1 0	IP-INC8	
JO = Jump on overflow	0 1 1 1 0 0 0 0	IP-INC8	
JS = Jump on sign	0 1 1 1 1 0 0 0	IP-INC8	
JNE/JNZ = Jump on not equal/not zer0	0 1 1 1 0 1 0 1	IP-INC8	
JNL/JGE = Jump on not less/greater or equal	0 1 1 1 1 1 0 1	IP-INC8	
JNLE/JG = Jump on not less or equal/greater	0 1 1 1 1 1 1 1	IP-INC8	
JNB/JAE = Jump on not below/above or equal	0 1 1 1 0 0 1 1	IP-INC8	
JNBE/JA = Jump on not below or equal/above	0 1 1 1 0 1 1 1	IP-INC8	
JNP/JPO = Jump on not par/par odd	0 1 1 1 1 0 1 1	IP-INC8	
JNO = Jump on not overflow	0 1 1 1 0 0 0 1	IP-INC8	

Figure 3-71 Continued.

CONTROL TRANSFER (Cont'd.)

RET = Return from CALL:	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
JNS = Jump on not sign	0 1 1 1 1 0 0 1	IP-INC8				
LOOP = Loop CX times	1 1 1 0 0 0 1 0	IP-INC8				
LOOPZ/LOOPE = Loop while zero/equal	1 1 1 0 0 0 0 1	IP-INC8				
LOOPNZ/LOOPNE = Loop while not zero/equal	1 1 1 0 0 0 0 0	IP-INC8				
JCXZ = Jump on CX zero	1 1 1 0 0 0 1 1	IP-INC8				

INT = Interrupt:					
Type specified	1 1 0 0 1 1 0 1	DATA-8			
Type 3	1 1 0 0 1 1 0 0				
INTO = Interrupt on overflow	1 1 0 0 1 1 1 0				
IRET = Interrupt return	1 1 0 0 1 1 1 1				

PROCESSOR CONTROL					
CLC = Clear carry	1 1 1 1 1 0 0 0				
CMC = Complement carry	1 1 1 1 0 1 0 1				
STC = Set carry	1 1 1 1 1 0 0 1				
CLD = Clear direction	1 1 1 1 1 1 0 0				
STD = Set direction	1 1 1 1 1 1 0 1				
CLI = Clear interrupt	1 1 1 1 1 0 1 0				
STI = Set interrupt	1 1 1 1 1 0 1 1				
HLT = Halt	1 1 1 1 0 1 0 0				
WAIT = Wait	1 0 0 1 1 0 1 1				
ESC = Escape (to external device)	1 1 0 1 1 x x x	m o d y y y r / m	(DISP-LO)	(DISP-HI)	
LOCK = Bus lock prefix	1 1 1 1 0 0 0 0				
SEGMENT = Override prefix	0 0 1 reg 1 1 0				

would be assembled as

00111101	10100010	00000101

instead of

10000001	11111000	10100010	00000101

Appendix: 8086/8088 Instruction Set Summary

The first two columns in Table A-1 give the mnemonic and a brief description of each instruction. The third column indicates the number of clock cycles needed to execute the instruction. For the 8086, four clock cycles need to be added for each word transfer with an odd address. For the 8088, four clock cycles need to be added for each word transfer. There are two numbers of clock cycles for the conditional branch and loop instructions; the one before the "/" is for when a branch is not taken and the other number is for when the branch is taken. The fourth column gives the number of bytes, or the range of numbers of bytes, for the instruction, and the fifth column indicates the possible flag settings using the following symbols:

-	Not affected
0	Cleared to 0
1	Set to 1
x	Set or cleared according to the result
u	Undefined
r	Restored from previously saved value

The sixth column contains the page number where the definition of the instruction can be found.

TABLE A-1 8086/8088 INSTRUCTION SET SUMMARY

Mnemonic	Description	Clock cycles	Number of bytes	Flags O D I T S Z A P C	Page ref.
AAA	ASCII adjust for addition	4	1	u - - - u u x u x	76
AAD	ASCII adjust for division	60	2	u - - - x x u x u	76
AAM	ASCII adjust for multiplication	83	1	u - - - x x u x u	76
AAS	ASCII adjust for subtraction	4	1	u - - - u u x u x	76
ADC	Add with carry			x - - - x x x x x	65
	Register to register	3	2		
	Memory to register	9+EA	2–4		
	Register to memory	16+EA	2–4		
	Immediate to register	4	3–4		
	Immediate to memory	17+EA	3–6		
	Immediate to accumulator	4	2–3		
ADD	Addition			x - - - x x x x x	65
	Register to register	3	2		
	Memory to register	9+EA	2–4		
	Register to memory	16+EA	2–4		
	Immediate to register	4	3–4		
	Immediate to memory	17+EA	3–6		
	Immediate to accumulator	4	2–3		
AND	Logical AND			0 - - - x x u x 0	93
	Register to register	3	2		
	Memory to register	9+EA	2–4		
	Register to memory	16+EA	2–4		
	Immediate to register	4	3–4		
	Immediate to memory	17+EA	3–6		
	Immediate to accumulator	4	2–3		
CALL	Call a procedure			- - - - - - - - -	157
	Intrasegment direct	19	3		
	Intrasegment indirect through register	16	2		
	Intrasegment indirect through memory	21+EA	2–4		
	Intersegment direct	28	5		
	Intersegment indirect	37+EA	2–4		
CBW	Convert byte to word	2	1	- - - - - - - - -	68
CLC	Clear carry flag	2	1	- - - - - - - - 0	92
CLD	Clear direction flag	2	1	- 0 - - - - - - -	92
CLI	Clear interrupt flag	2	1	- - 0 - - - - - -	92
CMC	Complement carry flag	2	1	- - - - - - - - x	92
CMP	Compare			x - - - x x x x x	69
	Register to register	3	2		
	Memory to register	9+EA	2–4		
	Register to memory	9+EA	2–4		
	Immediate to register	4	3–4		
	Immediate to memory	10+EA	3–6		

TABLE A-1 (*continued*)

Mnemonic	Description	Clock cycles	Number of bytes	Flags O D I T S Z A P C	Page ref.
	Immediate to accumulator	4	2–3		
CMPS/ CMPSB/ CMPSW	Compare string/ Compare byte string/ Compare word string			x - - - x x x x x	208
	Not repeated	22			
	Repeated	9+22/rep			
CWD	Convert word to double word	5	1	- - - - - - - - - -	68
DAA	Decimal adjust for addition	4	1	u - - - x x x x x	74
DAS	Decimal adjust for subtraction	4	1	u - - - x x x x x	74
DEC	Decrement by 1			x - - - x x x x -	69
	16-bit register	2	1		
	8-bit register	3	2		
	Memory	15+EA	2–4		
DIV	Unsigned division			u - - - u u u u u	70
	8-bit register	80–90	2		
	16-bit register	144–162	2		
	8-bit memory	(86–96) +EA	2–4		
	16-bit memory	(150–168) +EA	2–4		
ESC	Escape			- - - - - - - - - -	457
	Register	2	2		
	Memory	8+EA	2–4		
HLT	Halt	2	1	- - - - - - - - - -	91
IDIV	Integer division			u - - - u u u u u	70
	8-bit register	101–112	2		
	16-bit register	165–184	2		
	8-bit memory	(107–118) +EA	2–4		
	16-bit memory	(171–190) +EA	2–4		
IMUL	Integer multiplication			x - - - u u u u x	70
	8-bit register	80–98	2		
	16-bit register	128–154	2		
	8-bit memory	(86–104) +EA	2–4		
	16-bit memory	(134–160) +EA	2–4		
IN	Input from I/O port			- - - - - - - - - -	232
	Fixed port	10	2		
	Variable port	8	1		
INC	Increment by 1			x - - - x x x x -	69
	16-bit register	2	1		
	8-bit register	3	2		
	Memory	15+EA	2–4		
INT	Interrupt			- - 0 0 - - - - -	172

TABLE A-1 (*continued*)

Mnemonic	Description	Clock cycles	Number of bytes	Flags O D I T S Z A P C	Page ref.
	Type = 3	52	1		
	Type ≠ 3	51	2		
INTO	Interrupt if overflow		1	- - 0 0 - - - - -	172
	Interrupt is taken	53			
	Interrupt is not taken	4			
IRET	Return from interrupt	24	1	r r r r r r r r r	172
JA/	Jump if above/	16/4	2	- - - - - - - - -	80
JNBE	Jump if not below or equal				
JAE/	Jump if above or equal/	16/4	2	- - - - - - - - -	80
JNB/	Jump if not below/				
JNC	Jump if not carry				
JB/	Jump if below/	16/4	2	- - - - - - - - -	80
JNAE/	Jump if not above or equal/				
JC	jump if carry				
JBE/	Jump if below or equal/	16/4	2	- - - - - - - - -	80
JNA	Jump if not above				
JCXZ	Jump if CX is zero	18/6	2	- - - - - - - - -	88
JE/	Jump if equal/	16/4	2	- - - - - - - - -	80
JZ	Jump if zero				
JG/	Jump if greater/	16/4	2	- - - - - - - - -	80
JNLE	Jump if not less or equal				
JGE/	Jump if greater or equal/	16/4	2	- - - - - - - - -	80
JNL	Jump if not less				
JL/	Jump if less/	16/4	2	- - - - - - - - -	80
JNGE	Jump if not greater or equal				
JLE/	Jump if less or equal/	16/4	2	- - - - - - - - -	80
JNG	Jump if not greater				
JMP	Jump			- - - - - - - - -	84
	Intrasegment direct short	15	2		
	Intrasegment direct	15	3		
	Intersegment direct	15	5		
	Intrasegment indirect through memory	18+EA	2–4		
	Intrasegment indirect through register	11	2		
	Intersegment indirect	24+EA	2–4		
JNE/	Jump if not equal/	16/4	2	- - - - - - - - -	80
JNZ	Jump if not zero				
JNO	Jump if not overflow	16/4	2	- - - - - - - - -	80
JNP/	Jump if not parity/	16/4	2	- - - - - - - - -	80
JPO	Jump if parity odd				
JNS	Jump if not sign	16/4	2	- - - - - - - - -	80
JO	Jump if overflow	16/4	2	- - - - - - - - -	80
JP/	Jump if parity/	16/4	2	- - - - - - - - -	80
JPE	Jump if parity even				
JS	Jump if sign	16/4	2	- - - - - - - - -	80
LAHF	Load AH from flags	4	1	- - - - - - - - -	92
LDS	Load pointer using DS	16+EA	2–4	- - - - - - - - -	60
LEA	Load effective address	2+EA	2–4	- - - - - - - - -	60

TABLE A-1 (*continued*)

Mnemonic	Description	Clock cycles	Number of bytes	Flags O D I T S Z A P C	Page ref.
LES	Load pointer using ES	16+EA	2–4	- - - - - - - - - -	60
LOCK	Lock bus	2	1	- - - - - - - - - -	455
LODS/	Load string/		1	- - - - - - - - - -	208
LODSB/	Load byte string/				
LODSW	Load word string				
	Not repeated	12			
	Repeated	9+13/rep			
LOOP	Loop	17/5	2	- - - - - - - - - -	88
LOOPE/	Loop if equal/	18/6	2	- - - - - - - - - -	88
LOOPZ	Loop if zero				
LOOPNE/	Loop if not equal/	19/5	2	- - - - - - - - - -	88
LOOPNZ	Loop if not zero				
MOV	Move			- - - - - - - - - -	60
	Accumulator to memory	10	3		
	Memory to accumulator	10	3		
	Register to register	2	2		
	Memory to register	8+EA	2–4		
	Register to memory	9+EA	2–4		
	Immediate to register	4	2–3		
	Immediate to memory	10+EA	3–6		
	Register to SS, DS, or ES	2	2		
	Memory to SS, DS, or ES	8+EA	2–4		
	Segment register to register	2	2		
	Segment register to memory	9+EA	2–4		
MOVS/	Move string/		1	- - - - - - - - - -	208
MOVSB/	Move byte string/				
MOVSW	Move word string				
	Not repeated	18			
	Repeated	9+17/rep			
MUL	Unsigned multiplication			x - - - u u u u x	70
	8-bit register	70–77	2		
	16-bit register	118–133	2		
	8-bit memory	(76–83) +EA	2–4		
	16-bit memory	(124–139) +EA	2–4		
NEG	Negate			x - - - x x x x x	69
	Register	3	2		
	Memory	16+EA	2–4		
NOP	No operation	3	1	- - - - - - - - - -	91
NOT	Logical NOT			- - - - - - - - - -	93
	Register	3	2		
	Memory	16+EA	2–4		
OR	Logical OR			0 - - - x x u x 0	93
	Register to register	3	2		
	Memory to register	9+EA	2–4		
	Register to memory	16+EA	2–4		

TABLE A-1 (*continued*)

Mnemonic	Description	Clock cycles	Number of bytes	Flags O D I T S Z A P C	Page ref.
	Immediate to accumulator	4	2–3		
	Immediate to register	4	3–4		
	Immediate to memory	17 + EA	3–6		
OUT	Output to I/O port			- - - - - - - - -	232
	Fixed port	10	2		
	Variable port	8	1		
POP	Pop word off stack			- - - - - - - - -	153
	Register	8	1		
	Segment register SS, DS, or ES	8	1		
	Memory	17 + EA	2–4		
POPF	Pop flags off stack	8	1	r r r r r r r r r	153
PUSH	Push word onto stack			- - - - - - - - -	153
	Register	11	1		
	Segment register	10	1		
	Memory	16 + EA	2–4		
PUSHF	Push flags onto stack	10	1	- - - - - - - - -	153
RCL	Rotate left through carry			x - - - - - - - x	98
	Register with single-shift	2	2		
	Register with variable-shift	8 + 4/bit	2		
	Memory with single-shift	15 + EA	2–4		
	Memory with variable-shift	20 + EA + 4/bit	2–4		
RCR	Rotate right through carry			x - - - - - - - x	98
	Register with single-shift	2	2		
	Register with variable-shift	8 + 4/bit	2		
	Memory with single-shift	15 + EA	2–4		
	Memory with variable-shift	20 + EA + 4/bit	2–4		
REP	Repeat string operation	2	1	- - - - - - - - -	213
REPE/	Repeat operation while equal/	2	1	- - - - - - - - -	213
REPZ	Repeat operation while zero				
REPNE/	Repeat operation while not equal/	2	1	- - - - - - - - -	213
REPNZ	Repeat operation while not zero				
RET	Return from procedure			- - - - - - - - -	157
	Intrasegment	8	1		
	Intrasegment with constant	12	3		
	Intersegment	18	1		
	Intersegment with constant	17	3		
ROL	Rotate left			x - - - - - - - x	98
	Register with single-shift	2	2		

TABLE A-1 (*continued*)

Mnemonic	Description	Clock cycles	Number of bytes	Flags O D I T S Z A P C	Page ref.
	Register with variable-shift	8+4/bit	2		
	Memory with single-shift	15+EA	2–4		
	Memory with variable-shift	20+EA +4/bit	2–4		
ROR	Rotate right			x - - - - - - - x	98
	Register with single-shift	2	2		
	Register with variable-shift	8+4/bit	2		
	Memory with single-shift	15+EA	2–4		
	Memory with variable-shift	20+EA +4/bit	2–4		
SAHF	Store AH into flags	4	1	- - - - r r r r r	92
SAL/	Shift arithmetic left/			x - - - x x u x x	98
SHL	Shift logical left				
	Register with single-shift	2	2		
	Register with variable-shift	8+4/bit	2		
	Memory with single-shift	15+EA	2–4		
	Memory with variable-shift	20+EA +4/bit	2–4		
SAR	Shift arithmetic right			x - - - x x u x x	98
	Register with single-shift	2	2		
	Register with variable-shift	8+4/bit	2		
	Memory with single-shift	15+EA	2–4		
	Memory with variable-shift	20+EA +4/bit	2–4		
SBB	Subtract with borrow			x - - - x x x x x	65
	Register from register	3	2		
	Memory from register	9+EA	2–4		
	Register from memory	16+EA	2–4		
	Immediate from accumulator	4	2–3		
	Immediate from register	4	3–4		
	Immediate from memory	17+EA	3–6		
SCAS/	Scan string/		1	x - - - x x x x x	208
SCASB/	Scan byte string/				
SCASW	Scan word string				
	Not repeated	15			
	Repeated	9+15/rep			
SHR	Shift logical right			x - - - x x u x x	98
	Register with single-shift	2	2		
	Register with variable-shift	8+4/bit	2		
	Memory with single-shift	15+EA	2–4		
	Memory with variable-shift	20+EA +4/bit	2–4		
STC	Set carry flag	2	1	- - - - - - - - 1	92
STD	Set direction flag	2	1	- 1 - - - - - - -	92

TABLE A-1 *(continued)*

Mnemonic	Description	Clock cycles	Number of bytes	Flags O D I T S Z A P C	Page ref.
STI	Set interrupt flag	2	1	- - 1 - - - - - -	92
STOS/	Store string/		1	- - - - - - - - -	208
STOSB/	Store byte string/				
STOSW	Store word string				
	Not repeated	11			
	Repeated	9 + 10/rep			
SUB	Subtraction			x - - - x x x x x	65
	Register from register	3	2		
	Memory from register	9 + EA	2–4		
	Register from memory	16 + EA	2–4		
	Immediate from accumulator	4	2–3		
	Immediate from register	4	3–4		
	Immediate from memory	17 + EA	3–6		
TEST	Test			0 - - - x x u x 0	93
	Register with register	3	2		
	Memory with register	9 + EA	2–4		
	Immediate with accumulator	4	2–3		
	Immediate with register	5	3–4		
	Immediate with memory	11 + EA	3–6		
WAIT	Wait while $\overline{\text{TEST}}$ pin not asserted	3 + 5n	1	- - - - - - - - -	457
XCHG	Exchange			- - - - - - - - -	60
	Register with accumulator	3	1		
	Register with memory	17 + EA	2–4		
	Register with register	4	2		
XLAT/ XLATB	Translate	11	1	- - - - - - - - -	221
XOR	Logical exclusive OR			0 - - - x x u x 0	93
	Register with register	3	2		
	Memory with register	9 + EA	2–4		
	Register with memory	16 + EA	2–4		
	Immediate with accumulator	4	2–3		
	Immediate with register	4	3–4		
	Immediate with memory	17 + EA	3–6		

Uffenbeck, John The 8086/8088 family: design, programming, and interfacing Prentice-Hall 1987. pp.x + 630.
ISBN 0 13 246638 4
Reprint pp 45-107

Copied pursuant to S53B of
the Copyright Act as amended
on behalf of Murdoch Uni.
on 17.2.89

Week 7

2.3 ADDRESSING MODES

From Chap. 1 we learned that a computer instruction is made up of an operation code and zero, one, or two operands. The op-code identifies the operation to be performed while the operands identify the source and destination of the data operated on. The operands can specify a CPU register, a memory location in one of the memory segments, or an I/O port. The different ways in which the microprocessor generates these operand addresses are called the *addressing modes*. Table 2.2 lists the nine modes available with the 8086/88 microprocessor.

Immediate Addressing Mode

Instructions that use the immediate addressing mode get their data as part of the instruction. This mode represents a special case in which the data (located within the instruction) is stored in the code segment instead of the data segment. Note the *object code* column in Table 2.2. The data for the MOV AX,1000H instruction is supplied immediately after the instruction op-code B8.

TABLE 2.2 ADDRESSING MODES OF THE 8086/88 MICROPROCESSOR

Addressing mode	Object code	Coding example			
		Mnemonic	Segment for memory access	Symbolic operation	Description
Immediate	B8 00 10	MOV AX,1000H	Code	AH ← 10H; AL ← 00	Source of data is within instruction
Register	8B D1	MOV DX,CX	Within CPU	DX ← CX	Source and destination of data are CPU registers
Direct	8A 26 00 10	MOV AH,[MEMBDS][a]	Data	AH ← [1000H]	Memory address is supplied within the instruction
Register indirect	8B 04	MOV AX,[SI]	Data	AL ← [SI]; AH ← [SI + 1]	Memory address is supplied in an index or pointer register
	FF 25	JMP [DI]	Data	IP ← [DI + 1:DI]	
	FE 46 00	INC BYTE PTR[BP][b]	Stack	[BP] ← [BP] + 1	
	FF 0F	DEC WORD PTR[BX][b]	Data	[BX + 1:BX] ← [BX + 1:BX] − 1	
Indexed	8B 44 06	MOV AX,[SI + 6][c]	Data	AL ← [SI + 6]; AH ← [SI + 7]	Memory address is the sum of the index register plus a displacement within the instruction
	FF 65 06	JMP [DI + 6][c]	Data	IP ← [DI + 7:DI + 6]	
Based	8B 46 02	MOV AX,[BP + 2][c]	Stack	AL ← [BP + 2]; AH ← [BP + 3]	Memory address is the sum of the BX or BP base registers plus a displacement within instruction
	FF 67 02	JMP [BX + 2][c]	Data	IP ← [BX + 3:BX + 2]	

Based and indexed	8B 00	MOV AX,[BX + SI]	Data	AL ← [BX + SI] AH ← [BX + SI + 1]	Memory address is the sum of an index register and a base register
	FF 21	JMP [BX + DI]	Data	IP ← [BX + DI + 1:BX + DI]	
	FE 02	INC BYTE PTR[BP + SI][b]	Stack	[BP + SI] ← [BP + SI] + 1	
	FF 0B	DEC WORD PTR[BP + DI][b]	Stack	[BP + DI + 1:BP + DI] ← [BP + DI + 1:BP + DI] − 1	
Based and indexed with displacement	8B 40 05	MOV AX,[BX + SI + 5][c]	Data	AL ← [BX + SI + 5] AH ← [BX + SI + 6]	Memory address is the sum of an index register, a base register, and a displacement within instruction
	FF 61 05	JMP [BX + DI + 5][c]	Data	IP ← [BX + DI + 6:BX + DI + 5]	
	FE 42 05	INC BYTE PTR[BP + SI + 5][b,c]	Stack	[BP + SI + 5] ← [BP + SI + 5] + 1	
	FF 4B 05	DEC WORD PTR[BP + DI + 5][b,c]	Stack	[BP + DI + 6:BP + DI + 5] ← [BP + DI + 6:BP + DI + 5] − 1	

Strings	A4	MOVSB	Extra,Data	[ES:DI] ← [DS:SI]
				If DF = 0, then
				SI ← SI + 1; DI ← DI + 1
				If DF = 1, then
				SI ← SI − 1; DI ← DI − 1
					The memory source address is register SI in the data segment, and the memory destination address is register DI in the extra segment

[a]MEMBDS is assumed to point at location 1000H in the data segment. The brackets are optional.

[b]BYTE PTR and WORD PTR avoid the ambiguity of byte or word access.

[c]The displacement is added to the pointer or base register as a 2's-complement signed binary number.

You might also note in the object code column that the data is supplied *backward*, with the most significant byte first (this is very important if you are hand assembling your programs).

The immediate addressing mode is commonly used to load a register or memory location with some initial data. Following instructions then operate on this data. The one restriction is that the segment registers may not be specified. In the following section we show how to get around this problem.

Register Addressing Mode

Some instructions simply operate on or move data back and forth between CPU registers. MOV DX,CX puts a copy of register CX in register DX. Similarly, INC BH adds 1 to the contents of register BH. Note that the operand requires no memory reference.

The register addressing mode can be combined with the immediate addressing mode to load a segment register: for example,

```
MOV   AX,8010H      ;Load AX with 8010H
MOV   DS,AX         ;Copy AX into DS
```

The result is to move 8010H into register DS.

Direct Addressing Mode

All of the remaining addressing modes listed in Table 2.2 require an access to memory for at least one of the instruction operands. In the direct addressing mode the memory address is supplied directly as part of the instruction. However, as mentioned at the end of Sec. 2.2, labels are often given to these addresses so that the programmer need not be concerned with the numeric address. The instruction MOV AH,MEMBDS has the word interpretation "Store the contents of memory location MEMBDS in register AH."[1]

As note a in Table 2.2 indicates, MEMBDS is presumed to have been defined as a memory byte location in the data segment—in this specific case, address 1000H. You can see this address as the last two bytes in the object code column.

Indirect Memory Access Addressing Modes

The direct addressing mode is useful for infrequent memory accesses. However, when a memory location must be read from or written to several times within a program, the repeated fetching of the two-byte logical address makes this mode inefficient.

The indirect addressing modes solve this problem by storing the memory address in a pointer or index register (BX, BP, SI, or DI). In addition, a 2's-complement displacement can be added to the pointer or index to offset the location pointed at.[2] Table 2.3 shows the various combinations possible.

[1]Because MEMBDS is presumed to have been defined with the DB operator, the assembler knows that this label represents the contents of a memory location, not a constant.

[2]In App. A we review 2's-complement numbers.

TABLE 2.3 EFFECTIVE ADDRESS CALCULATIONS FOR THE INDIRECT MEMORY ACCESS ADDRESSING MODES

	Effective address				
Addressing mode	Displacement		Base register		Index register
Register indirect	None	+	BX or BP	+	None
	None	+	None	+	SI or DI
Indexed	+127 to −128	+	None	+	SI or DI
Based	+127 to −128	+	BX or BP	+	None
Based and indexed	None	+	BX or BP	+	SI or DI
Based and indexed with displacement	+127 to −128	+	BX or BP	+	SI or DI

In general, a displacement can be added to a base register and the result added to an index register. This is the "deluxe" mode, called "based and indexed with displacement." In between are the "economy" modes, which use only a base or index register with or without a displacement. The resulting address is often referred to as the *effective address* (EA).

Note that the displacement is limited to a single byte, allowing the EA to be varied +127 bytes (7FH) or −128 bytes (80H) from the pointer base. Probably the most powerful feature of these indirect addressing modes is that the memory address can be changed based on program conditions. For example, we might set BX to point to the base of a table, perform a calculation in register SI to determine an index into this table, and finally test a flag to determine an offset value into this indexed table. This technique is explored in Chap. 4.

Sometimes you will see the indirect addressing mode operands written as in the instruction MOV AX,TABLES[SI], where TABLE is a label given to some memory location (presumably the base of a data table). This instruction can also be written MOV AX,[TABLE+SI]. However, the first form suggests that SI is used as an *index* into an array called TABLE. Thus if SI = 5, the fifth element in the TABLE array is accessed. Most assemblers will accept either form of the instruction.

Example 2.5

Identify the addressing mode for each of the following instructions:

```
(a) MOV AH,47H
(b) MOV AH,[BP+2]
(c) MOV AH,[BP+SI]
(d) MOV AH,[XRAY]       ;XRAY defines a memory location
(e) MOV AH,TEMP[BX]     ;TEMP defines a memory location
```

Solution (a) Immediate; (b) based; (c) based and indexed; (d) direct; (e) based.

When studying Table 2.2 you probably noticed two operators that we have not yet mentioned: BYTE PTR and WORD PTR. These are assembler pseudo operators needed to help the assembler determine if a byte or word memory access is required. For example, does the instruction INC [BP] mean "increment the memory *word*" or "increment the

memory *byte* pointed to by register BP''? You cannot tell. By using the BYTE PTR or WORD PTR prefix, the proper instruction codes will be generated by the assembler.

When the instruction specifies two operands, the register operand will determine the byte or word access. For example, MOV AX,[BP]—move to register AX the contents of memory pointed to by register BP—indicates the word pointed at by BP because register AX (a word register) is indicated as the destination.

Finally, note that the default memory segments for all the indirect addressing modes are the stack segment when register BP is involved, and the data segment when register BX, SI, or DI is involved.

String Addressing Mode

In ''computerese'' a *string* is a sequence of bytes or words stored in memory. Word processors store their text as long strings of ASCII characters. A data table is another string example. Because of their importance, the 8086/88 has several instructions designed specifically for handling strings of characters.

These instructions have a special addressing mode of their own and use DS:SI to point to the string source and ES:DI to point to the string destination. MOVSB moves the source data byte to the destination location. Note that SI and DI are automatically incremented or decremented depending on the value of DF. In Sec. 2.5 we cover all the string instructions in detail.

SELF-REVIEW 2.3 (answers on page 108)

2.3.1. An instruction operand may reference a _____, a _____, or an _____.

2.3.2. The instruction MOV BH,23H uses the _____ addressing mode.

2.3.3. The instruction DEC BYTE PTR[BX+2] uses the _____ addressing mode and decrements the _____ pointed to by register BX+2.

2.3.4. When using the _____ addressing mode, the memory address is supplied as part of the instruction.

2.3.5. MEMBDS is an example of a _____ given to a memory location.

2.3.6. What is meant by NOTE[BP]?

2.3.7. Write the mnemonic for the instruction required to move the word pointed at by register SI offset by 27D to register CX.

2.3.8. Which memory segment will be accessed for the word described in question 2.3.7?

2.3.9. Which 8086/88 general-purpose registers cannot be used as base pointer or index registers in the indirect memory access addressing modes?

2.4 DATA TRANSFER INSTRUCTIONS

Now that we are familiar with the 8086/88 programming model and the various addressing modes, it is time to turn our attention to the instruction set itself. You should realize that this is quite a job. There are over 3000 different 8086/88 instructions to consider if each op-code is combined with each addressing mode!

What I would like you to do is consider the remaining sections of this chapter to be a first "tour" through this complex instruction set. We will look at the various instructions organized into logical groups and presented in table form. These tables will then serve as a reference for you as we begin to assemble increasingly more complex programs in Chaps. 3 and 4.

The data transfer group of instructions presented in this section allow the microprocessor to communicate with the outside world via the input and output instructions. They also provide the means for moving data into and out of memory and between the CPU registers.

MOV Instruction

Table 2.4 lists the various types of data transfers possible using the MOV instruction. Note that the operand is always written in the form *destination,source*. The data moved can be a byte or a word, although the pointer, index, and segment registers can only be accessed as words. The destination and source cannot both specify memory locations. Note that the flags are unaffected by this group of instructions.

Many more combinations of the MOV instruction are possible than are shown in Table 2.4: for example,

```
MOV  AX,[BX]
MOV  AX,[BX+4]
MOV  AX,[SI]
MOV  AX,[SI-6]
MOV  AX,[BP+SI]
MOV  AX,[BP+SI+3EH]
```

These are all memory-to-register instructions but use the various forms of the memory indirect addressing modes.

Example 2.6

Assume that memory (byte) location COUNT holds the number of counts that a certain operation is to be repeated. Write an 8086/88 program to load CH with 00 and CL with the contents of COUNT.

Solution

```
MOV   CH,0        ;Move 0 to register CH
MOV   CL,COUNT    ;Copy the contents of COUNT
                  ;to register CL
```

Special Data Transfer Instructions

There are other data transfer instructions that do not use the MOV mnemonic. These are shown in Table 2.5. The XCHG instruction performs a register-to-register or register-to-memory swap that takes the place of three MOV instructions.

The 8 low-order flag bits can be stored in or loaded from register AH. These flags

TABLE 2.4 MOV INSTRUCTION[a]

Addressing mode	Object code	Coding example: Mnemonic	Segment for memory access	Symbolic operation	Description
MOV *dest,source*	8B C3	MOV AX,BX	Within CPU	AX ← BX	Register to register
	8A E3	MOV AH,BL	Within CPU	AH ← BL	Register to register
	A1 00 10	MOV AX,MEMWDS[b]	Data	AL ← [1000H] AH ← [1001H]	Memory to register
	A0 02 10	MOV AL,MEMBDS[c]	Data	AL ← [1002H]	Memory to register
	89 1E 00 10	MOV MEMWDS,BX[b]	Data	[1000H] ← BL [1001H] ← BH	Register to memory
	88 1E 02 10	MOV MEMBDS,BL[c]	Data	[1002H] ← BL	Register to memory
	C7 06 00 10 34 12	MOV MEMWDS,1234H[b]	Data	[1000H] ← 34H	Immediate data to

C6 06 02 10 34	MOV MEMBDS,34H[c]	Data	[1002H] ← 34H	Immediate data to memory
B0 10	MOV AL,10H	Code	AL ← 10H	Immediate data to register
B8 00 10	MOV AX, 1000H	Code	AL ← 00H; AH ← 10H	Immediate data to register
8E D8	MOV DS,AX	Within CPU	DS ← AX	General register to segment register
8C C2	MOV DX,ES	Within CPU	DX ← ES	Segment register to general register
8E 06 00 10	MOV ES,MEMWDS[b]	Data	ES ← [1001H:1000H]	Memory to segment register
8C 0E 00 10	MOV MEMWDS,CS[b]	Data	[1001H:1000H] ← CS	Segment register to memory

[a]The memory location can be specified using any of the addressing modes shown in Table 2.2. In this table the direct addressing mode is shown for all memory access examples.

[b]MEMWDS is assumed to point at the word beginning at location 1000H in the data segment.

[c]MEMBDS is assumed to point at the byte at location 1002H in the data segment.

TABLE 2.5 SPECIAL DATA TRANSFER INSTRUCTIONS[a]

General mnemonic Op-code	Operand	Object code	Coding example: Mnemonic	Segment for memory access	Symbolic operation	Description
XCHG	*dest,source*	93	XCHG AX,BX	Within CPU	AX⇆BX	Switch the contents of the word or byte source operand with the destination operand; none of the flags are affected.
		86 C7	XCHG AL,BH	Within CPU	AL⇆BH	
		87 14	XCHG [SI],DX	Data	[SI]⇆DL; [SI + 1]⇆DH	
LAHF		9F	LAHF	Within CPU	AH ← Flagsl	Copy the low-order flag byte into AH
SAHF		9E	SAHF	Within CPU	flagsl ← AH	Copy AH into the low-order flag byte
IN	*accumulator,port*	E4 26	IN AL,26H	[b]	AL ← port 26H	Input a byte or word from direct I/O ports 0–255
		E5 26	IN AH,26H	[b]	AL ← port 26H; AH ← port 27H	
		EC	IN AL,DX	[b]	AL ← port DX	Input a byte or word from indirect I/O ports 0–65535; the port address is in DX; none of the flags are affected
		ED	IN AX,DX	[b]	AL ← port DX; AH ← port DX + 1	
OUT	*port,accumulator*	E6 26	OUT 26H,AL	[b]	port 26H ← AL	Output a byte or word to direct I/O ports 0–255
		E7 26	OUT 26H,AX	[b]	port 26H ← AL; port 27H ← AH	

		EE	OUT DX,AL	[b]	port DX ← AL	Output a byte or word to indirect I/O ports 0–65535; the port address is in DX; none of the flags are affected
		EF	OUT DX,AX	[b]	port DX ← AL; port DX + 1 ← AH	
LEA	*dest,source*[c]	8D 1E 00 10	LEA BX,MEMBDS[c]	Data	BL ← 00; BH ← 10H	The effective address of the source operand is transferred to the destination operand; none of the flags are affected
LDS	*dest,source*[d]	C5 1C	LDS BX,DWORD PTR[SI]	Data	BL ← [SI]; BH ← [SI + 1]; DS ← [SI + 3:SI + 2]	Transfer a 32-bit pointer variable from the source operand in memory to the destination register and register DS or ES; none of the flags are affected
LES	*dest,source*[d]	C4 1C	LES BX,DWORD PTR[SI]	Data	BL ← [SI]; BH ← [SI + 1]; ES ← [SI + 3:SI + 2]	
XLAT		D7	XLAT	Data	AL ← [BX + AL]	Replace the byte in AL with a byte from the 256-byte table beginning at [BX]; use AL as an offset into this table; none of the flags are affected

[a]Memory operands may be accessed using any of the addressing modes shown in Table 2.2. In this table the register indirect mode is used for example only.

[b]The IN and OUT instructions do not involve the segment registers.

[c]MEMBDS is assumed to point at the byte at location 1000H in the data segment.

[d]The destination must be a 16-bit CPU register.

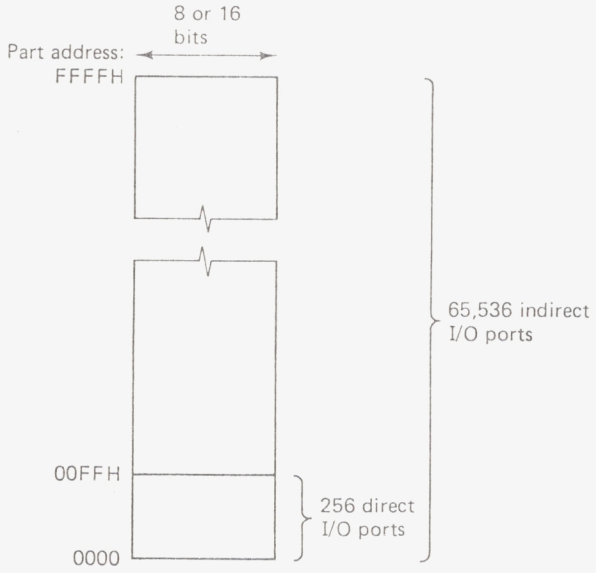


Figure 2.10 The 8086/88 can access up to 65,536 different 8- or 16-bit I/O ports.

are in the same position as those in the 8-bit 8080 and 8085 microprocessors. Thus LAHF and SAHF are useful when translating programs from these processors to the 8086/88.

The IN and OUT instructions allow the 8086/88 to transfer data between itself and the outside world. A byte or word of data can be input or output but must pass through the accumulator. AL must be used as the source or destination for 8-bit I/O operations.

The I/O device address, also called the *port* address, can range from 0 to FFFFH. This allows up to 65,536 different I/O ports. The port address can be specified in two ways. In the *direct mode*, the instruction supplies the address but is limited to ports 0–255 (i.e., one byte). In the *indirect mode*, register DX holds the 16-bit port address, allowing access to all 65,536 ports. This is shown in Fig. 2.10.

Example 2.7

Write the 8086/88 program required to output the word in BX to I/O ports 8004H and 8005H.

Solution

```
MOV   DX,8004H      ;Point DX at the port address
MOV   AX,BX         ;Data must be in AX
OUT   DX,AX         ;Output the data
```

All the indirect memory access addressing modes require a pointer or index register to hold the memory address. This register can be loaded using the immediate addressing mode as

```
MOV   BX,1000H
```

but this requires the programmer to know the address of the target memory location—violating one of the goals of assembly language programming. A better technique is to

use the LEA (load effective address) instruction. For example, assume that address 1000H has been given the label MEMBDS. Then the instruction

```
LEA  BX,MEMBDS
```

will load register BX with the EA of MEMBDS—in this case 1000H. The point is, we would not have to know this address; the assembler would look it up and insert it as part of the LEA object code.

In some cases an entirely new address, including the segment register, must be determined. The LDS (load pointer using DS) and LES (load pointer using ES) instructions are intended for this purpose. They load the 16-bit destination register and ES or DS segment register with the contents of the double-word memory operand.

Example 2.8

Assume that MEMWWDS defines a double word beginning at address 1000H in the data segment as shown in Fig. 2.11. What physical address will BX be pointing to after the following instruction sequence?

```
LEA  SI,MEMWWDS
LDS  BX,DWORD PTR[SI]
```

Solution The first instruction loads SI with the address of MEMWWDS—1000H. The second instruction loads BX and DS with the contents of the double word pointed at by SI. In this case BX = 8010H and DS = E000H. Now because BX defaults to the data segment, the physical address pointed to by BX becomes

$$E0000H + 8010H = E8010H$$

Note: The single instruction LDS BX,MEMWWDS could be used with the same effect.

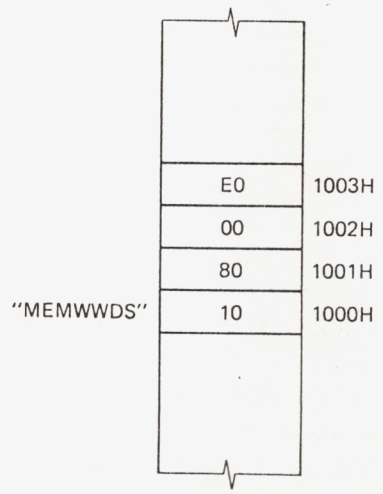


Figure 2.11 Memory organization for Ex. 2.8.

The last instruction in Table 2.5 is XLAT and is useful for extracting data from a table using register AL as the offset into the table. In Chap. 4 we provide an example of the use of this special instruction.

Segment Override

Every instruction that accesses memory has a particular segment register that it uses to determine the EA of the memory operand. Table 2.1 has shown the default segment register assignments.

The alternate segment column in this table indicates that the default assignment can be overridden for some types of memory references. Table 2.6 shows examples of the four segment override prefix instructions CS:, DS:, ES:, and SS:. Note in the object code column that the effect is to precede the normal object codes with the single byte 2E, 26, 3E, or 36. The segment override is in effect for the one instruction only and thus must be specified for each instruction to be overridden.

A typical application for the segment override is to allow data to be stored in the code segment. For example, consider the instruction sequence

```
CODE        SEGMENT
COUNT       DB    OFFH
MOV  AL,CS:COUNT
. . . (program continues)
```

The variable COUNT is defined within the code segment, requiring the MOV instruction to use the CS: override to access this memory location. Some assembler programs are smart enough to insert the segment override prefix on their own. If COUNT is known to the assembler to be in the code segment (due to the DB operator), the correct prefix will be generated automatically. Using CS: does not hurt, however, and serves as a check that COUNT is really in the code segment.

There is another way to store variables in the code segment—and that is to define CS = DS.

Example 2.9

Explain the operation of the following program:

```
MOV  CX,CS          ;Copy CS into CX
MOV  DS,CX          ;Copy CS into DS
MOV  AL,COUNT       ;Load AL with the contents of COUNT
```

Solution The first two instructions make DS = CS. The last instruction loads AL with the contents of COUNT in the data segment. However, because DS = CS, COUNT is actually in the same physical memory space as the program. No segment override is needed.

Because a segment override is required to access data in the code segment, programs that use this technique must pay a penalty of one extra byte of code per data access instruction. For this reason, you are encouraged to store program data in the data segment. The resulting programs will require fewer bytes and execute faster.

TABLE 2.6 SEGMENT OVERRIDE PREFIX

General mnemonic Op-code	Operand	Object code	Coding example: Mnemonic	Segment for memory access	Symbolic operation	Description
	CS:	2E A1 00 10 2E 89 4E 00	MOV AX,CS:MEMWCS[a] MOV CS:[BP],CX	Code Code	AX ← CS:[1001H:1000H] CS:[BP] ← CX	The default segment for the memory source or destination operand is overriden to become the code segment
	ES:	26 A1 00 10 26 89 4E 00	MOV AX,ES:MEMWES[a] MOV ES:[BP],CX	Extra Extra	AX ← ES:[1001H:1000H] ES:[BP] ← CX	The default segment for the memory source or destination operand is overridden to become the extra segment
	DS:	3E 89 4E 00	MOV DS:[BP],CX	Data	DS:[BP] ← CX	The default segment for the memory source or destination operand is overridden to become the data segment
	SS:	36 A1 00 10 36 89 0F	MOV AX,SS:MEMWSS[a] MOV SS:[BP],CX	Stack Stack	AX ← SS:[1001H:1000H] SS:[BP] ← CX	The default segment for the memory source or destination operand is overridden to become the extra segment

[a]The assembler will automatically generate the segment override if these memory words have previously been defined to be in the appropriate segment. Each word is assumed to begin at address 1000H in the defined segment.

F-REVIEW 2.4 *(answers on page 108)*

2.4.1. Using the MOV instruction, which of the following data transfers is illegal? (a) memory to memory; (b) memory to register; (c) register to memory; (d) segment register to memory.

2.4.2. Write the mnemonic for the instruction to load memory location MEMWDS with the contents of the DX register.

2.4.3. The 8086/88 can access _____ indirect I/O ports using register _____ to hold the port address.

2.4.4. Data can be input or output through any of the 8086/88 general-purpose registers. (True/False)

2.4.5. Write the mnemonic for the instruction to load register DI with the base address of a data table starting at location TABLE.

2.4.6. Instruction op-codes are normally fetched from the code segment but can be overridden and fetched from the stack, data, or extra segments. (True/False)

STRING INSTRUCTIONS

As mentioned in Sec. 2.3, the 8086/88 has several instructions for moving large blocks of data or *strings*. Table 2.7 lists the five string instructions. For all these instructions the memory source is DS:SI and the memory destination is ES:DI. Segment overrides can be applied only to the source address—the destination must be the extra segment.

The offset memory pointers, SI and DI, are automatically incremented or decremented depending on the state of DF—by 1 for bytes or by 2 for words. Because the instructions use dedicated registers defining the source and destination of data, the destination operand is not required (note that the object code is the same for both forms). Using the operand does make the assembly listing easier to read, however.

The STOS (store string byte or word) and LODS (load string byte or word) instructions transfer a byte or word from the accumulator to memory or from memory to the accumulator. The MOVS (move string byte or word) instruction combines these two operations, transferring the byte or word from the memory source to the memory destination.

The SCAS (scan string byte or word) and CMPS (compare string byte or word) instructions allow the destination byte or word to be compared with the accumulator (SCAS) or the memory source (CMPS). After execution the flags are set to reflect the relationship of the destination element to the source element. The conditional jump instructions can then be used to make decisions such as "Jump if AL is greater than the memory byte" or "Jump if the destination memory word equals the source memory word."

Repeat Prefix

You might be wondering how the string instructions are used to operate on large blocks of data. So far we have seen only the ability to handle single bytes or words. This is where the REP (repeat) prefix shown in Table 2.8 comes in.

Preceding the string instructions STOS or MOVS with the REP instruction causes

TABLE 2.7 STRING INSTRUCTIONS

General mnemonic		Object code	Coding example			
Op-code	Operand		Mnemonic	Segment for memory access	Symbolic operation	Description
STOSB	None	AA	STOSB	Extra	ES:[DI] ← AL If DF = 0, DI ← DI + 1 If DF = 1, DI ← DI − 1	Transfer a byte or word from register AL or AX to the string element addressed by DI in the extra segment; if DF = 0, increment DI, else decrement DI; the flags are not affected
STOSW	None	AB	STOSW	Extra	ES:[DI] ← AL ES:[DI + 1] ← AH If DF = 0, DI ← DI + 2 If DF = 1, DI ← DI − 2	
STOS	*dest*[a]	AA	STOS MEMBES	Extra	Same as STOSB if DI = MEMBES	
		AB	STOS MEMWES	Extra	Same as STOSW if DI = MEMWES	
LODSB		AC	LODSB	Data	AL ← DS:[SI] If DF = 0, SI ← SI + 1 If DF = 1, SI ← SI − 1	Transfer a byte or word from the string element addressed by SI in the data segment to register AL or AX; if DF = 0, increment SI, else decrement SI; the flags are not affected
LODSW		AD	LODSW	Data	AL ← DS:[SI] AH ← DS:[SI + 1] If DF = 0, SI ← SI + 2 If DF = 1, SI ← SI − 2	
LODS	*source*[a]	AC	LODS MEMBDS	Data	Same as LODSB if SI = MEMBDS	
		AD	LODS MEMWDS	Data	Same as LODSW if SI = MEMWDS	

TABLE 2.7 (continued)

General mnemonic Op code	Operand	Object code	Coding example: Mnemonic	Segment for memory access	Symbolic operation	Description
MOVSB	None	A4	MOVSB	Data, extra	ES:[DI] ← DS:[SI] If DF = 0, DI ← DI + 1 SI ← SI + 1 If DF = 1, DI ← DI − 1 SI ← SI − 1	Transfer a byte or word from the string element addressed by SI in the data segment to the string element addressed by DI in the extra segment; if DF = 0, increment SI and DI, else decrement SI and DI; the flags are not affected
MOVSW	None	A5	MOVSW	Data, extra	ES:[DI] ← DS:[SI] ES:[DI + 1] ← DS:[SI + 1] If DF = 0, DI ← DI + 2 SI ← SI + 2 If DF = 1, DI ← DI − 2 SI ← SI − 2	
MOVS	*dest,source*[a]	A4	MOVS MEMBES, MEMBDS	Extra, data	Same as MOVSB if SI = MEMBDS and DI = MEMBES	
		A5	MOVS MEMWES, MEMWDS	Extra, data	Same as MOVSW if SI = MEMWDS and DI = MEMWES	
SCASB		AE	SCASB	Extra	AL − ES:[DI]; update flags If DF = 0, DI ← DI + 1 If DF = 1, DI ← DI − 1	Subtract the byte or word of the string element addressed by DI in the extra segment from AL or AX; if DF = 0, increment DI, else decrement DI; the flags are updated to reflect the relationship of the destination operand to the source operand
SCASW		AF	SCASW	Extra	AX − ES:[DI + 1:DI]; update flags If DF = 0, DI ← DI + 2 If DF = 1, DI ← DI − 2	
SCAS	*dest*[a]	AE	SCAS MEMBES	Extra	Same as SCASB if DI = MEMBES	

		AF	SCAS MEMWES	Extra	Same as SCASW if DI = MEMWES	
CMPSB		A6	CMPSB	Extra, data	DS:[SI] − ES:[SI]; update flags If DF = 0, DI ← DI + 1 SI ← SI + 1 If DF = 1, DI ← DI − 1 SI ← SI − 1	Subtract the byte or word of the destination string element addressed by DI in the extra segment from the byte or word of the source string element addressed by SI in the data segment; if DF = 0, increment DI and SI, else decrement SI and DI; the flags are updated to reflect the relationship of the destination operand to the source operand
CMPSW		A7	CMPSW	Extra, data	DS:[SI + 1:SI] − ES:[DI + 1:DI]; update flags If DF = 0, DI ← DI + 2 SI ← SI + 2 If DF = 1, DI ← DI − 2 SI ← SI − 2	
CMPS	*dest,source*[a]	A6	CMPS MEMBES, MEMBDS	Extra, data	Same as CMPSB if SI = MEMBDS and DI = MEMBES	
		A7	CMPS MEMWES, MEMWDS	Extra, data	Same as CMPSW if SI = MEMWDS and DI = MEMWES	

[a]The destination or source operand must identify the string element as a byte of a word. Specifying the operand makes the instructions intent clear; however, the object code is the same as the nonoperand form.

TABLE 2.8 REP PREFIX

General mnemonic		Object code	Coding example			
Op-code	Operand		Mnemonic	Segment for memory access	Symbolic operation	Description
REP		F3 AA	REP STOSB	Extra	STOSB; CX ← CX − 1 Repeat until CX = 0	The string instruction following the REP prefix is repeated until CX is decremented to 0
		F3 AB	REP STOSW	Extra	STOSW; CX ← CX − 1 Repeat until CX = 0	
		F3 A4	REP MOVSB	Extra, data	MOVSB; CX ← CX − 1 Repeat until CX = 0	
		F3 A5	REP MOVSW	Extra, data	MOVSW; CX ← CX − 1 Repeat until CX = 0	
REPE/REPZ[a]		F3 AE	REPZ SCASB	Extra	SCASB; CX ← CX − 1 Repeat if ZF = 1 and CX ≠ 0	Repeat the string operation if the scan or compare is equal (i.e., ZF = 1) *and* CX ≠ 0; decrementing CX does not affect the flags
		F3 AF	REPZ SCASW	Extra	As above except SCASW	
		F3 A6	REPZ CMPSB	Extra, data	As above except CMPSB	
		F3 A7	REPZ CMPSW	Extra, data	As above except CMPSW	
REPNE/REPNZ[a]		F2 AE	REPNE SCASB	Extra	SCASB: CX ← CX − 1 Repeat if ZF = 0 and CX ≠ 0	Repeat the string operation if the scan or compare is not equal (i.e., ZF = 0) *and* CX ≠ 0; decrementing CX does not affect the flags
		F2 AF	REPNE SCASW	Extra	As above except SCASW	
		F2 A6	REPNE CMPSB	Extra, data	As above except CMPSB	
		F2 A7	REPNE CMPSW	Extra, data	As above except CMPSW	

[a]Either form of the memonic can be used.

these instructions to be repeated a number of times equal to the contents of the CX register. By loading CX with the number of words or bytes to be moved, a single string instruction (and REP prefix) can move up to 65,536 bytes.

Example 2.10

Write the 8086/88 program required to fill the 1000D-byte block of memory in the extra segment beginning at address BLOCK with the data byte 20H (ASCII space).

Solution

```
MOV   AL,20H        ;AL holds the data byte
LEA   DI,BLOCK      ;DI holds the address of BLOCK
MOV   CX,03E7H      ;Load CX with 1000D
REP   STOSB         ;Store AL at ES:DI, increment
                    ;DI, and repeat 1000 times
```

Note: DF is assumed to be 0.

The REPE/REPZ (repeat while equal or zero) and REPNE/REPNZ (repeat while not equal or not zero) forms of the REP prefix are intended for use with the SCAS and CMPS string instructions. They allow the string operation to be repeated while equal (REPE/REPZ) or while not equal (REPNE/REPNZ). These can be taken advantage of when performing a table-lookup operation or testing to ensure that two strings are equal.

SELF-REVIEW 2.5 (answers on page 108)

2.5.1. A block of sequential characters stored in memory is also called a _____.

2.5.2. When it is desired to move a block of data from one location in memory to another, the _____ instruction should be used with the REP prefix.

2.5.3. When DF = 1 the LODSW instruction will automatically _____ register _____ by _____.

2.5.4. Under what two conditions will the REPNE CMPSB instructions pass control to the following instructions?

2.6 LOGICAL INSTRUCTIONS

The logical instructions refer not to those instructions that seem logical (then the others are illogical?) but to the Boolean logic functions, such as AND, OR, NOT, exclusive-OR, and to the rotate and shift instructions. These instructions are all performed in the arithmetic-logic unit and usually affect all the flags.

Boolean Functions

Table 2.9 lists the five logic functions available with the 8086/88 microprocessor. Each function is performed bit by bit between the source and destination operands. An example will help illustrate this concept.

TABLE 2.9 LOGICAL INSTRUCTIONS

General mnemonic		Object code	Coding example			
Op-code	Operand		Mnemonic[a]	Segment for memory access	Symbolic operation	Description
NOT	*dest*	F7 D3	NOT BX	Within CPU	BX ← $\overline{BX}$	Complement all bits of the byte or word operand; none of the flags are affected
		F6 14	NOT BYTE PTR[SI]	Data	[SI] ← $\overline{[SI]}$	
AND	*dest,source*	23 CA	AND CX,DX	Within CPU	CX ← CX · DX	Perform a bit-by-bit AND of the source and destination byte or word operands storing the result in the destination operand; AF is undefined, all other flags are updated[b]
		22 3C	AND BH,BYTE PTR[SI]	Data	BH ← BH · [SI]	
		25 00 80	AND AX,8000H	Code	AX ← AX · 8000H	
OR	*dest, source*	0B CA	OR CX,DX	Within CPU	CX ← CX + DX	Perform a bit-by-bit OR of the source and destination byte or word operands storing the result in the destination operand; AF is undefined, all other flags are updated[b]
		0A 3C	OR BH,BYTE PTR[SI]	Data	BH ← BH + [SI]	
		0D 00 80	OR AX,8000H	Code	AX ← AX + 8000H	

XOR	*dest,source*	33 CA	XOR CX,DX	Within CPU	CX ← CX ⊕ DX	Perform a bit-by-bit exclusive-OR of the source and destination byte or word operands storing the result in the destination operand; AF is undefined, all other flags are updated[b]
		32 3C	XOR BH,BYTE PTR[SI]	Data	BH ← BH ⊕ [SI]	
		35 00 80	XOR AX,8000H	Code	AX ← AX ⊕ 8000H	
TEST	*dest,source*	85 D1	TEST CX,DX	Within CPU	CX · DX; update flags	Perform a bit-by-bit AND of the source and destination byte or word operands; the operands remain unchanged; AF is undefined, all other flags are updated[b]
		84 3C	TEST BH,BYTE PTR[SI]	Data	BH · [SI]; update flags	
		A9 00 80	TEST AX,8000H	Code	AX · 8000H; update flags	

[a]Any of the addressing modes can be used.

[b]CF and 0F are reset.

Example 2.11

Determine the contents of register AL and the state of the flags after the following instructions are executed.

```
MOV  AL,6DH     ;Load AL with 6DH
MOV  BH,40H     ;Load BH with 40H
AND  AL,BH      ;AND AL with BH
```

Solution Write the register contents in binary:

```
 01101101  (AL)
·01000000  (BH)
 01000000 = 40H  (AL)
```

The flags are affected as follows:

```
CF = 0    ;CF is reset by the AND instruction
PF = 0    ;40H has an odd number of logic 1's
AF = X    ;AF is undefined
ZF = 0    ;The result is not zero
SF = 0    ;Bit 7 is reset
OF = 0    ;OF is reset by the AND instruction
```

The AND operation has several applications. As pointed out in Ex. 2.11, selected bits of the destination operand can be forced low by choosing those bits as 0 in the source operand. Another application is to consider the source operand a *mask* for testing selected bits of the destination operand. In Ex. 2.11 all bits are masked except bit 6. If this bit is 0, the result is 0; if it is a 1, the result is nonzero. The instruction sequence

```
AND  AL,BH
JZ   START
```

will transfer control to memory location START if bit 6 of register AL is a 0 (assuming that BH = 40H).

Using the AND instruction as a mask results in a "destructive" bit test because the contents of the destination operand is altered by the instruction. The TEST instruction, however, performs the same function but does not alter the source or destination operands. The conditional transfer to location START can be rewritten

```
TEST  AL,40H
JZ    START
```

This is a particularly handy instruction to use when several bits must be tested. If the first test fails, the second bit can be tested, and so on.

The OR instruction can be used to force selected bits high. For example, OR AL,80H will force bit 7 of AL high without changing any of the other bits in this register. Similarly, the exclusive-OR function can be used to *complement* selected bits. For example, XOR AL,80H will complement bit 7 of register AL without changing any of the other bits.

TABLE 2.10 SHIFT AND ROTATE INSTRUCTIONS

General mnemonic			Coding example[a]			
Op-code	Operand	Object code	Mnemonic	Segment for memory access	Symbolic operation	Description
SAL/SHL[b]	*dest,count*	D1 E0	SAL AX,1	Within CPU	See Fig. 2.12	Shift word or byte operand left or right once or CL times; AF is undefined, all other flags are updated; for single-bit shifts, OF is set if the sign of the operand changes
		D3 E0	SAL AX,CL	Within CPU	See Fig. 2.12	
SAR	*dest,count*	D0 F8	SAR AL,1	Within CPU	See Fig. 2.12	
SHR	*dest,count*	D2 F8	SAR AL,CL	Within CPU	See Fig. 2.12	
		D1 2C	SHR WORD PTR[SI],1	Data	See Fig. 2.12	
		D2 2C	SHR BYTE PTR[SI],CL	Data	See Fig. 2.12	
RCL	*dest,count*	D1 D3	RCL BX,1	Within CPU	See Fig. 2.12	Rotate word or byte operand left or right once or CL times; only CF and OF are affected; for single-bit rotates, OF is set if the sign of the operand changes
		D3 D3	RCL BX,CL	Within CPU	See Fig. 2.12	
RCR	*dest,count*	D0 DB	RCR BL,1	Within CPU	See Fig. 2.12	
		D2 DB	RCR BL,CL	Within CPU	See Fig. 2.12	
ROL	*dest,count*	D1 04	ROL WORD PTR[SI],1	Data	See Fig. 2.12	
		D2 04	ROL BYTE PTR[SI],CL	Data	See Fig. 2.12	
ROR	*dest,count*	D1 0E 00 10	ROR MEMWDS,1[c]	Data	See Fig. 2.12	
		D2 0E 04 10	ROR MEMBDS,CL[d]	Data	See Fig. 2.12	

[a]Any of the addressing modes can be used.

[b]Either form of the mnemonic can be used.

[c]MEMWDS is assumed to point at the word beginning at location 1000H in the data segment.

[d]MEMBDS is assumed to point at the byte at location 1004H in the data segment.

Shift and Rotate Instructions

There are several shift and rotate instructions; these are listed in Table 2.10 and diagrammed in Fig. 2.12. Be sure to notice that the rotated quantity can be an 8- or a 16-bit CPU register or memory location.

The main difference between a shift and a rotate is that the shifted bits "fall off" the end of the register, whereas the rotated bits "wrap around." Within the shift group of instructions there are both *arithmetic* (SAL and SAR) and *logical* (SHL and SHR) shift instructions.

The arithmetic shifts operate so that the sign bit (in bit 7 or 15) does not change

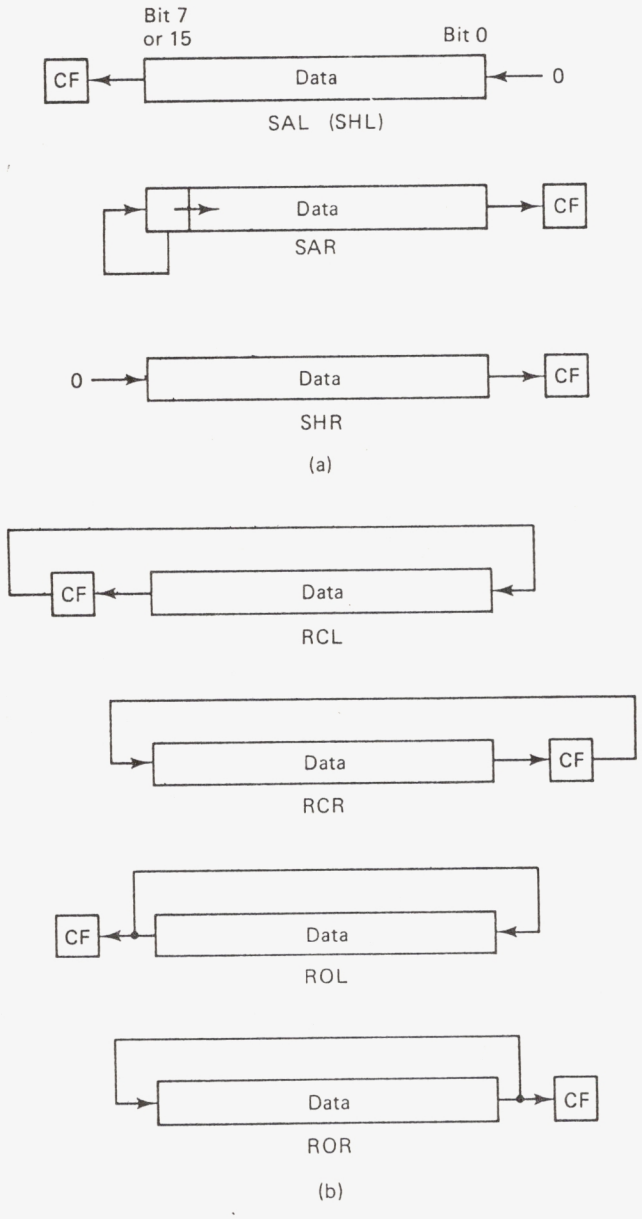


Figure 2.12 The 8086/88 has (a) three shift instructions and (b) four rotate instructions.

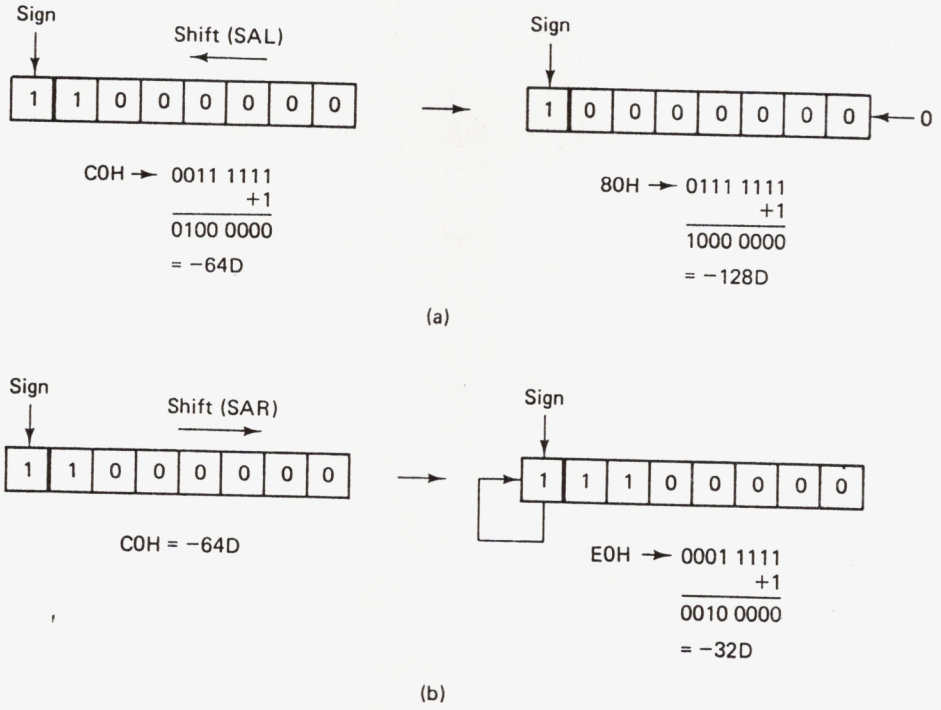


Figure 2.13 (a) The SAL instruction *multiplies* the signed binary number by a power of 2. (b) The SAR instruction *divides* the signed binary number by a power of 2.

when the shift occurs. Figure 2.13 shows the effect of the SAL and SAR instructions on the data byte COH. As you can see, the SAL instruction in effect multiplies the data by 2 (maintaining the correct sign), and SAR divides the data by 2. The overflow flag (OF) will be set if the shifted quantity exceeds the limits for an 8- or 16-bit register (+127 to −128 for bytes and +32767 to −32768 for words).

The shift and rotate instructions can be repeated up to 255 times by loading register CL with the desired count. For example, to rotate the contents of register DX left five times through the carry, the following instructions should be given:

```
MOV  CL,5
RCL  DX,CL
```

Example 2.12

Determine the contents of registers AX, BX, and CX after the following program is run.

```
MOV  CL,3
MOV  AX,7FH
MOV  BX,0505H
ROL  AX,CL
AND  AH,BH
OR   BL,AL
```

Solution The results are as follows.

		AX	BX	CX
MOV	CL,3	????	????	??03
MOV	AX,7FH	007F	????	??03
MOV	BX,0505H	007F	0505	??03
ROL	AX,CL	03F8	0505	??03
AND	AH,BH	01F8	0505	??03
OR	BL,AL	01F8	05FD	??03

SELF-REVIEW 2.6 (answers on page 108)

2.6.1. Which logical instruction should be used to force bits 0 and 1 of register DH low without changing any of the other bits?

2.6.2. Which logical instruction should be used to force bits 0 and 1 of register DH high without changing any of the other bits?

2.6.3. You need to determine if bit 3 of register AH is high without changing the contents of AH. Which instruction should be used?

2.6.4. What is the content of register BL after the following instructions are executed?

```
MOV  BL,0B2H
MOV  CL,2
SAR  BL,CL
```

2.6.5. Which two instructions will have the effect of dividing register BX by 8?

2.7 ARITHMETIC INSTRUCTIONS

The arithmetic instructions give the microprocessor its computational capabilities. But unlike earlier 8-bit processors, these instructions are not limited to addition and subtraction of 8-bit numbers in the accumulator. The 8086/88 can add and subtract 8- and 16-bit numbers in any of the general CPU registers, and using certain dedicated registers, perform multiplication and division of signed or unsigned numbers.

Addition and Subtraction Instructions

Examples of the various addition and subtraction instructions are given in Table 2.11. Note that the destination and source operands can be register and register, register and memory, memory and register, immediate data and register, or immediate data and memory.

There are two forms of addition and subtraction instructions. One includes the carry and the other does not. The ability to include the carry allows *multiple-precision* numbers to be handled. For example, suppose that we wish to add the 32-bit number in registers BX:AX to the 32-bit number in registers DX:CX storing the result in DX:CX. The addition would look as follows:

$$\begin{array}{rr} BX & AX \\ +DX & CX \\ \hline DX & CX \end{array}$$

Although there are no 32-bit addition instructions, the problem is easily solved using the ADC (add with carry) instruction.

```
ADD  CX,AX      ;CX ← CX + AX
ADC  DX,BX      ;DX ← DX + BX + CF
```

The first instruction does not include the carry, as there is no carry in at this point. If the addition of AX and CX sets CF, the second addition will add this to the sum of DX and BX.

The SUB (subtract) and SBB (subtract with borrow) instructions work similarly, with CF representing the borrow condition. When adding or subtracting one from a memory pointer or counter variable, the INC (increment) and DEC (decrement) instructions should be used. Note that the contents of a memory location can also be specified as the destination operand for these instructions.

The NEG (negate) instruction forms the 2's complement of the destination operand, effectively reversing its sign. This is done by subtracting the destination operand from 0.

Example 2.13

Determine the value of AL and the value of the flags following the instruction sequence

```
MOV  AL,5
NEG  AL
```

Solution AL → 00000000 − 00000101 = 11111011 = FBH = −5. The flags are affected as follows:

```
CF = 1    ;NEG sets CF except when the operand is 0
PF = 0    ;FBH has an odd number of logic 1's
AF = 1    ;There is a borrow out of bit 4
ZF = 0    ;The result is not 0
SF = 1    ;Bit 7 is set
OF = 0    ;There is no overflow condition
```

The CMP (compare) instruction is useful for determining the relative size of two operands. Normally, it is followed by a conditional jump instruction such as "jump if equal" or "jump if greater than or equal."

Multiplication and Division Instructions

Multiplication and division can be performed on signed or unsigned numbers as indicated in Table 2.12. The source operand can be a memory location or a CPU register, but the destination operand must be register AX (and DX for 32-bit results). The register usage

TABLE 2.11 ADDITION AND SUBTRACTION INSTRUCTIONS

General mnemonic			Coding example			
Op-code	Operand	Object code	Mnemonic	Segment for memory access	Symbolic operation	Description
ADD	*dest,source*	03 F2	ADD SI,DX	Within CPU	SI ← SI + DX	Replace the destination byte or word with the sum of the source and destination operands; update all flags
		00 2F	ADD BYTE PTR[BX],CH	Data	[BX] ← [BX] + CH	
		81 C7 00 80	ADD DI,8000H	Within CPU	DI ← DI + 8000H	
		81 06 00 10 00 80	ADD MEMWDS, 8000H[a]	Data	[1001H:1000H] ← [1001H:1000H] + 8000H	
ADC	*dest,source*	13 F2	ADC SI,DX	Within CPU	SI ← SI + DX + CF	Replace the destination byte or word with the sum of the source and destination operands plus the carry; update all flags
		10 2F	ADC BYTE PTR[BX],CH	Data	[BX] ← [BX] + CH + CF	
		81 D7 00 80	ADC DI,8000H	Within CPU	DI ← DI + 8000H + CF	
		81 16 00 10 00 80	ADC MEMWDS,8000H[a]	Data	[1001H:1000H] ← [1001H:1000H] + 8000H + CF	
SUB	*dest,source*	2B F2	SUB SI,DX	Within CPU	SI ← SI − DX	Replace the destination byte or word with the difference between the destination operand and source operand; update all flags
		28 2F	SUB BYTE PTR[BX],CH	Data	[BX] ← [BX] − CH	
		81 EF 00 80	SUB DI,8000H	Within CPU	DI ← DI − 8000H	
		81 2E 00 10 00 80	SUB MEMWDS,8000H[a]	Data	[1001H:1000H] ← [1001H:1000H] − 8000H	
SBB	*dest,source*	1B F2	SBB SI,DX	Within CPU	SI ← SI − DX − CF	Replace the destination byte or word with the difference between the destination operand and source operand plus carry; update all flags
		18 2F	SBB BYTE PTR[BX],CH	Data	[BX] ← [BX] − CH − CF	
		81 DF 00 80	SBB DI,8000H	Within CPU	DI ← DI − 8000H − CF	
		81 1E 00 10 00 80	SBB MEMWDS,8000H[a]	Data	[1001H:1000H] ← [1001H:1000H] − 8000H − CF	

INC *dest*[b]	FE C3	INC BL	Within CPU	BL ← BL + 1	Add one to the byte or word destination operand; store the result in the destination operand; all flags except CF are updated
	FF 05	INC WORD PTR[DI]	Data	[DI + 1:DI] ← [DI + 1:DI] + 1	
	FE 06 04 10	INC MEMBDS[c]	Data	[1004H] ← [1004H] + 1	
DEC *dest*[b]	FE CB	DEC BL	Within CPU	BL ← BL − 1	Subtract one from the byte or word destination operand, store the result in the destination operand; all flags except CF are updated
	FF 0D	DEC WORD PTR[DI]	Data	[DI + 1:DI] ← [DI + 1:DI] − 1	
	FE 0E 04 10	DEC MEMBDS[c]	Data	[1004H] ← [1004H] − 1	
NEG *dest*[b]	F6 DB	NEG BL	Within CPU	BL ← 0 − BL	Form the 2's complement of the byte or word destination operand; update all flags (CF = 1 except when the operand is zero)
	F7 1D	NEG WORD PTR[DI]	Data	[DI + 1:DI] ← 0 − [DI + 1:DI]	
	F6 1E 04 10	NEG MEMBDS[c]	Data	[1004H] ← 0 − [1004H]	
CMP *dest,source*	3A C4	CMP AL,AH	Within CPU	AL − AH; update flags	Subtract the byte or word source operand from the similar destination operand; the operands remain unchanged; the flags are updated to reflect the relationship of the destination operand to the source operand
	39 0D	CMP [DI],CX	Data	[DI + 1:DI] − CX; update flags	
	81 3E 00 10 00 80	CMP MEMWDS,8000H[a]	Data	[1001H:1000H] − 8000H; update flags	
	81 FF 00 80	CMP DI,8000H	Within CPU	DI − 8000H; update flags	

[a]MEMWDS is assumed to point at the word beginning at location 1000H in the data segment.

[b]Immediate operands are not allowed.

[c]MEMBDS is assumed to point at the byte at location 1004H in the data segment.

TABLE 2.12 MULTIPLICATION AND DIVISION INSTRUCTIONS

General mnemonic		Object code	Coding example			
Op-code	Operand[a]		Mnemonic	Segment for memory access	Symbolic operation	Description
MUL	*source*	F6 E3	MUL BL	Within CPU	AX ← AL*BL	Unsigned multiplication of the byte or word source operand and the accumulator; word results are stored in AX and double word results are stored in DX:AX (see Fig. 2.14); if the result cannot be stored in a single byte (for byte multiplication) or a single word (for word multiplication) CF and OF are set; cleared otherwise, all other flags are undefined
		F7 E1	MUL CX	Within CPU	DX:AX ← AX*CX	
			MUL BYTE PTR[BX]	Data	AX ← AL*[BX]	
		F6 27	MUL MEMWDS[b]	Data	DX:AX ← AX*[1001H:1000H]	
		F7 26 00 10				
IMUL	*source*	F6 EB	IMUL BL	Within CPU	AX ← AL*BL (signed)	Same as MUL except signed numbers are allowed; the source operand is limited to −128 to +127 for byte multiplication and −32768 to +32767 for word multiplication; CF and OF are set if the result cannot be represented in the low order register of the result; cleared otherwise, the sign is extended to the high order register; the other flags are not affected
		F7 E9	IMUL CX	Within CPU	DX:AX ← AX*CX (signed)	
		F6 2F	IMUL BYTE PTR[BX]	Data	AX ← AL*[BX] (signed)	
		F7 2E 00 10	IMUL MEMWDS[b]	Data	DX:AX ← AX*[1001H:1000H] (signed)	

DIV	*source*	F6 F3	DIV BL	Within CPU	AX ← AL/BL	Unsigned division of the accumulator (for byte divisors) or accumulator and DX (for word divisors); for byte divisors the result is returned in AL with the remainder in AH; for word divisors the result is returned in AX with remainder in DX (see Fig. 2.14); if the quotient exceeds the capacity of its destination register (AL or AX) a type 0 interrupt is generated; all flags are undefined
		F7 F1	DIV CX	Within CPU	DX:AX ← AX/CX	
		F6 37	DIV BYTE PTR[BX]	Data	AX ← AL/[BX]	
		F7 36 00 10	DIV MEMWDS[b]	Data	DX:AX ← AX/[1001H:1000H]	
IDIV	*source*	F6 FB	IDIV BL	Within CPU	AX ← AL/BL (signed)	Same as DIV except signed division is performed; the source operand is limited to −128 to +127 for byte division and −32768 to +32767 for word division
		F7 F9	IDIV CX	Within CPU	DX:AX ← AX/CX (signed)	
		F6 3F	IDIV BYTE PTR[BX]	Data	AX ← AL/[BX] (signed)	
		F7 3E 00 10	IDIV MEMWDS[b]	Data	DX:AX ← AX/[1001H:1000H] (signed)	

[a]Immediate operands are not allowed (except with the 80186).
[b]MEMWDS is assumed to point at the word beginning at location 1000H in the data segment.

is shown in Fig. 2.14 for both byte and word operations. Note that immediate operands are not allowed.

Example 2.14

Write a program to input two 8-bit unsigned numbers from input ports A0H and B0H and output the product to 16-bit output port 7080H.

Solution

```
IN    AL,0A0H     ;Get first number to AL
MOV   BL,AL       ;Save in BL
IN    AL,0B0H     ;Get second number to AL
MUL   BL          ;From product in AX
MOV   DX,7080H    ;Port address to DX
OUT   DX,AX       ;Output the product
```

Division can be performed on the word in AX or the double word in DX:AX. The divisor can be an 8- or a 16-bit memory location or CPU register. Note that a remainder is returned in register AH or DX when the result does not come out even.

Example 2.15

Write a program to divide the unsigned word input at indirect I/O port 8000H by 500D. Determine the result if the input data is 56,723.

Solution The program is as follows:

```
MOV   DX,8000H    ;DX holds port address
MOV   BX,01F4H    ;BX holds divisor (500D)
IN    AX,DX       ;Input word
DIV   BX          ;Calculate quotient in AX and
                  ;remainder in DX
```

The result is AX = INT(56,723/500) = 113 = 71H and DX = MOD(56,723/500) = 223 = 00DFH.[3]

A special problem can occur when the divisor is so small as to cause the quotient to overflow the register dedicated to the result. For example, dividing 65,000 by 2 leaves a result too large to fit in register AL alone. When this occurs a special *divide by zero* software interrupt is automatically generated by the 8086/88. This causes control to be transferred to the address stored in locations 00000–00003H. Section 2.8 includes a description of software interrupts.

The integer multiplication and division instructions are similar to the unsigned forms except that the most significant bit represents the sign of the number. For integer byte division the quotient is limited to the range +127 to −128 and to the range +32767 to −32768 for word division. IDIV and IMUL are often used with the CBW and CWD (convert byte to word and word to double word) instructions described in the next section.

[3]The MOD operator represents the remainder after the indicated division.

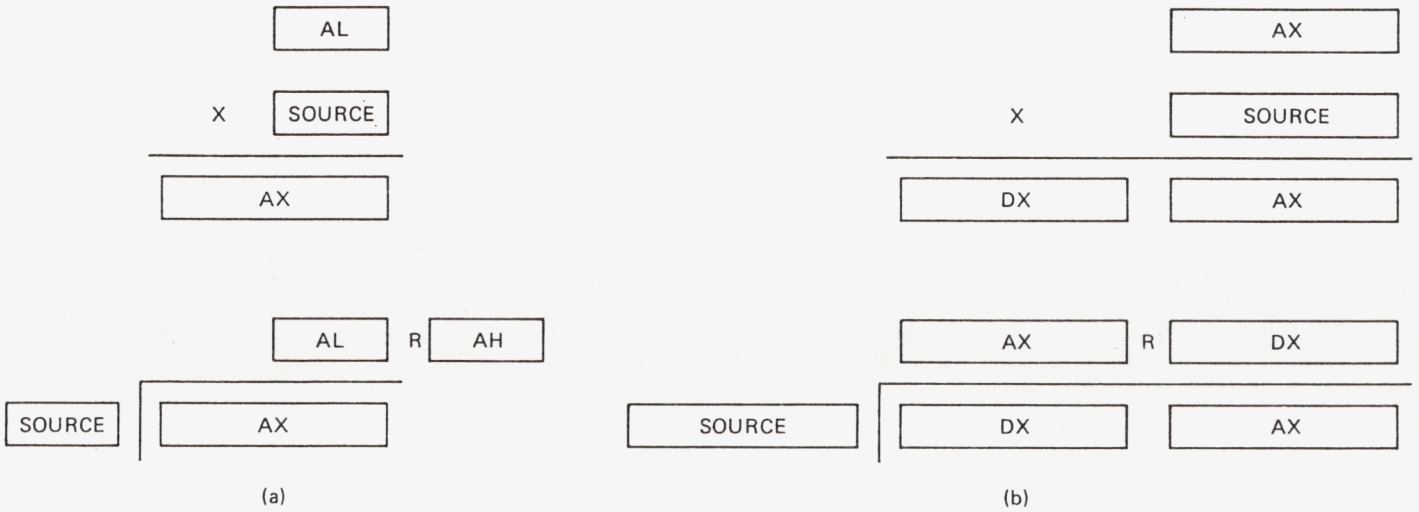


Figure 2.14 The multiplication and division instructions can be performed on (a) 8- or (b) 16-bit register or memory source operands. Specific registers are dedicated to these functions as shown. (From J. Uffenbeck, *Microcomputers and Microprocessors: The 8080, 8085, and Z-80*. Prentice-Hall, Englewood Cliffs, N.J., 1985.)

Arithmetic Adjust Instructions

In some cases the data to be operated on by the arithmetic instructions is encoded in BCD (decimal) or ASCII form. BCD encoding uses 4 bits to represent the decimal digits 0–9. The hex digits A–F are considered invalid. This means that one byte can hold two decimal digits—referred to as *packed decimal*. For example, the binary number 10011001 (99H) is the largest (one byte) packed decimal number allowed.

A decimal number can also be *unpacked* with one digit per byte. For example, 00001001 (09H) represents the unpacked decimal number nine.

An ASCII number is similar to an unpacked decimal number in that one byte can hold only one digit. In fact, it is very easy to convert an ASCII digit to decimal—simply subtract 30H. For example, the number 9 in ASCII is 39H (see Table 1.3). Subtracting 30H results in 09H − 9 in unpacked decimal form.

There are no special instructions for adding, subtracting, multiplying, or dividing these encoded numbers. This can lead to invalid results if care is not taken. For example, consider the following addition of two unpacked decimal numbers:

```
MOV  AL,7
MOV  BL,5
ADD  AL,BL
```

The result will be AL = 7 + 5 = 0CH (not 12!). The result is correct in binary form but invalid as a BCD number.

Fortunately, there are several arithmetic adjust instructions that can be used immediately before or after the arithmetic instructions to ensure valid decimal results. Table 2.13 lists these instructions.

In the preceding example, following the ADD AL,BL instruction with DAA—decimal adjust for addition—will correct the result so that AL = 12H. The DAS (decimal adjust for subtraction) instruction performs a similar function following the subtraction of two decimal operands.

When it is desired to convert the results of a decimal addition or subtraction to ASCII the AAA (ASCII adjust for addition) and AAS (ASCII adjust for subtraction) instructions should be used. However, they are to be used with unpacked decimal numbers only.

Note that the AAA and AAS instructions do not actually convert the result to ASCII form. However, the result in AL (and AH if a carry or borrow occurred) will be a valid unpacked decimal number. Adding 30H will convert this number to ASCII. For example, the decimal addition program becomes

```
MOV  AH,0          ;Be sure AH is 0
MOV  AL,7          ;First number
MOV  BL,5          ;Second number
ADD  AL,BL         ;AX = 000CH
AAA                ;AX = 0102
ADD  AX,3030H      ;AX = 3132H
```

TABLE 2.13 ARITHMETIC ADJUST INSTRUCTIONS

General mnemonic		Object code	Coding example			
Op-code	Operand		Mnemonic	Segment for memory access	Symbolic operation	Description
DAA	none	27	DAA	Within CPU	If AL · 0F > 9 or AF = 1, then AL ← AL + 6; AF ← 1 If AL > 9F or CF = 1, then AL ← AL + 60H; CF ← 1	Adjust AL to a pair of valid packed decimal digits following the addition of two valid packed or unpacked decimal operands; all flags except OF (undefined) are affected
DAS	none	2F	DAS	Within CPU	If AL · 0F > 9 or AF = 1, then AL ← AL − 6; AF ← 1 If AL > 9F or CF = 1, then AL ← AL − 60H; CF ← 1	Adjust AL to a pair of valid packed decimal digits following the subtraction of two valid packed or unpacked decimal operands; all flags except OF (undefined) are affected
AAA	none	37	AAA	Within CPU	If AL · 0F > 9 or AF = 1, then AL ← AL + 6; AH ← AH + 1; AF ← 1; CF ← AF; AL ← AL · 0F	Adjust AL to a single unpacked decimal number following the addition of two valid unpacked decimal operands; the high-order half-byte of AL is zeroed and AH is incremented by 1; all flags except AF and CF are undefined

00 0E AL = AL+6
AL = 14
61 0E AH = AH+1 = 01

TABLE 2.13 (continued)

General mnemonic		Object code	Coding example			
Op-code	Operand		Mnemonic	Segment for memory access	Symbolic operation	Description
AAS	none	3F	AAS	Within CPU	If AL · 0F > 9 or AF = 1, then AL ← AL − 6; AH ← AH − 1; AF ← 1; CF ← AF; AL ← AL · 0F	Adjust AL to a single unpacked decimal number following the subtraction of two valid unpacked decimal operands; the high-order half-byte of AL is zeroed and AH is decremented by 1; all flags except AF and CF are undefined
AAM	none	D4 0A	AAM	Within CPU	AH ← AL/0AH AL ← remainder	Following the multiplication of two valid unpacked decimal operands, AAM converts the result in AL to two valid unpacked decimal digits in AH and AL; all flags except PF, SF, and ZF are undefined

AAD	none	D5 0A	AAD	Within CPU	AL ← (AH * 0AH) + AL AH ← 0	Before dividing AX by a single-digit unpacked decimal operand, AAD converts the two-digit unpacked decimal number in AX to a binary number in AL and 0 in AH; the quotient produced by the following division will then be a valid unpacked decimal number in AL and remainder in AH; all flags except PF, SF, and ZF are undefined
CBW	none	98	CBW	Within CPU	If AL < 80H, then AH ← 0 If AL > 7F, then AH ← FFH	Before dividing AX by a byte operand, CBW extends the sign of a byte dividend in AL into AH, thus converting AL into a valid signed word in AX; none of the flags are affected
CWD	none	99	CWD	Within CPU	If AX < 8000H, then DX ← 0 If AX > 7FFFH, then DX ← FFFFH	Same as CBW but extends the sign of a word dividend in AX into a double word in DX:AX; none of the flags are affected

TABLE 2.14 JUMP INSTRUCTIONS

General mnemonic		Object code	Coding example			
Op-code	Operand		Mnemonic	Segment for memory access	Symbolic operation	Description
JMP	*near target*	E9 — — [a]	JMP MEMN[b]	Code	IP ← MEMN	Transfer control to (near) target location within the segment; the addressing mode may be direct, memory indirect, or register indirect
		FF 26 00 10	JMP [MEMWDS][c]	Data	IP ← [MEMWDS + 1:MEMWDS]	
		FF 27	JMP [BX]	Data	IP ← [BX + 1:BX]	
		FF E0	JMP AX	Within CPU	IP ← AX	
JMP	SHORT *target*	EB — [d]	JMP SHORT MEMS[e]	Code	IP ← MEMS	Transfer control to the (short) target location; there are no indirect forms of this instruction
JMP	*far target*	EA 03 00 D3 9E	JMP FAR PTR MEMF[f]	Code	IP ← 0003H; CS ← 9ED3H	Transfer control to the far target location outside the segment
		FF 2E 05 10	JMP [MEMWWDS][g]	Data	IP ← [1006H:1005H]; CS ← [1008H:1007H]	
		FF 2F	JMP DWORD PTR[BX]	Data	IP ← [BX + 1:BX]: CS ← [BX + 3:BX + 2]	

Jcond[h] *short target*	73 — [d]	JNC MEMS	Code	If CF = 0, then IP ← MEMS	Transfer control to the short target address if the condition is true; all conditional jumps require short targets
JCXZ *short target*	E3 — [d]	JCXZ MEMS	Code	If CX = 0, then IP ← MEMS	Transfer control to the short target address if CX = 0

[a]These two bytes are the 2's-complement offset between near memory location MEMN and the instruction immediately following the JMP instruction. For example, if the jump is to be 12 locations forward, the offset is 00 0C. Similarly, 12 locations backward requires an offset of FFF4. The offset is limited to 32767 locations forward (7FFFH) and −32768 locations backward (8000H).

[b]MEMN is assumed to point to a near-memory location (i.e., within the segment).

[c]MEMWDS is assumed to point at the word beginning at location 1000H in the data segment. The brackets are optional because MEMWDS defines a word, not a location to jump to.

[d]This byte is the 2's-complement offset between short memory location MEMS and the instruction immediately following the jump SHORT instruction. The offset is limited to 127 locations forward (7FH) and 128 locations backward (80H).

[e]MEMS is assumed to point to a short memory location (i.e., +127 or −128 locations from the address of the instruction immediately following the JMP SHORT instruction). The SHORT operator is optional if MEMS is already known by the assembler.

[f]MEMF is assumed to point to the far memory location 9ED3H:0003H.

[g]MEMWWDS is assumed to point to the double word beginning at location 1005H in the data segment. The brackets are optional because MEMWWDS defines a double word, not a location to jump to.

[h]See Table 2.15 for a list of testable conditions.

TABLE 2.16 LOOP INSTRUCTIONS

General mnemonic			Coding example			
Op-code	Operand	Object code	Mnemonic	Segment for memory access	Symbolic operation	Description
LOOP	*short target*	E2 —[a]	LOOP MEMS [b]	Code	CX ← CX − 1 If CX ≠ 0, then IP ← MEMS	Decrement CX and transfer control to the short target address if CX ≠ 0
LOOPE/[c] LOOPZ	*short target*	E1 —[a]	LOOPZ MEMS [b]	Code	CX ← CX − 1 If (CX ≠ 0)·(ZF = 1), then IP ← MEMS	Decrement CX and transfer control to the short target address if CX ≠ 0 AND the last instruction to affect the flags resulted in zero (ZF = 1)
LOOPNE/[c] LOOPNZ	*short target*	E0 —[a]	LOOPNZ MEMS[b]	Code	CX ← CX − 1; If (CX ≠ 0)·(ZF = 0), then IP ← MEMS	Decrement CX and transfer control to the short target address if CX ≠ 0 AND the last instruction to affect the flags resulted in a nonzero result (ZF = 0)

[a]This byte is the 2's-complement offset between short memory location MEMS and the instruction immediately following the LOOP instruction. The offset is limited to 127 locations forward (7FH) and 128 locations backward (80H).

[b]MEMS is assumed to point to a short memory location (i.e., +127 or −128 locations from the address of the instruction immediately following the LOOP instruction).

[c]Either form of the mnemonic may be used.

TABLE 2.17 PUSH AND POP INSTRUCTIONS

Addressing mode	Object code	Coding example: Mnemonic op-code operand	Segment for memory access	Symbolic operation	Description
PUSH *source*[a]	51	PUSH CX	Stack	SP ← SP − 2; [SP + 1] ← CH; [SP] ← CL	Decrement SP by 2 and transfer the word from the source operand to the top of the stack now pointed to by SP
	1E	PUSH DS	Stack	SP ← SP − 2; [SP + 1:SP] ← DS	
	FF 75 02	PUSH [DI + 2]	Stack, data	SP ← SP − 2; [SP + 1] ← [DI + 3]; [SP] ← [DI + 2]	
POP *dest*[a]	59	POP CX	Stack	CL ← [SP]; CH ← [SP + 1]; SP ← SP + 2	Transfer the word from the top of the stack pointed to by SP to the destination operand; increment SP by 2
	1F	POP DS	Stack	DS ← [SP + 1:SP]; SP ← SP + 2	
	8F 45 02	POP [DI + 2]	Data, stack	[DI + 3] ← [SP + 1]; [DI + 2] ← [SP]; SP ← SP + 2	
PUSHF none	9C	PUSHF	Stack	SP ← SP − 2; [SP + 1:SP] ← flags	Push the 16-bit flag word onto the stack
POPF none	9D	POPF	Stack	flags ← [SP + 1:SP]; SP ← SP + 2	Pop the top of the stack into the 16-bit flag word

[a]Immediate operands are not allowed (except with the 80186).

TABLE 2.18 CALL AND RETURN INSTRUCTIONS

General mnemonic		Object code	Coding example			
Op-code	Operand		Mnemonic	Segment for memory access	Symbolic operation	Description
Call	*near target*	E8 — —[a]	CALL MEMN[b]	Code	SP ← Sp − 2; [SP + 1:SP] ← IP; IP ← MEMN	IP is pushed onto the top of the stack and control is transferred within the segment to the near target address
		FF 16 00 10	CALL [MEMWDS][c]	Data	SP ← SP − 2; [SP + 1:SP] ← IP IP ← [1001H:1000H]	
		FF 15	CALL [DI]	Data	SP ← SP − 2; [SP + 1:SP] ← IP; IP ← [DI + 1:DI]	
		FF D7	CALL DI	Within CPU	SP ← SP − 2 [SP + 1:SP] ← IP; IP ← DI	
CALL	*far target*	9A 00 10 D3 09	CALL FAR PTR MEMF[d]	Code	SP ← SP − 2; [SP + 1:SP] ← CS; CS ← 09D3H; SP ← SP − 2; [SP + 1:SP] ← IP; IP ← 1000H	CS and IP are pushed onto the top of the stack and control is transferred to the new segment and far target address
		FF 1E 00 10	CALL [MEMWWDS][e]	Data	Same as above except: CS ← [1003H:1002H]; IP ← [1001H:1000H]	
		FF 1D	CALL DWORD PTR[DI]	Data	Same as above except: CS ← [DI + 3:DI + 2]; IP ← [DI + 1:DI]	

TABLE 2.18 (continued)

General mnemonic		Object code	Coding example			
Op-code	Operand		Mnemonic	Segment for memory access	Symbolic operation	Description
RET	*n* (near)[f]	C3	RET	Stack	IP ← [SP + 1:SP]; SP ← SP + 2	The word at the top of the stack is popped into IP transferring control to this new address; RET is normally used to return control to the instruction following a near subroutine call; if included, the optional pop value (n) is added to SP
		C2 08 00	RET 8	Stack	IP ← [SP + 1:SP]; SP ← SP + 2 + 8	
RET	*n* (far)[f]	CB	RET	Stack	IP ← [SP + 1:SP]; SP ← SP + 2; CS ← [SP + 1:SP]; SP ← SP + 2	As above except that the double word at the top of the stack is popped into IP and CS transferring control to this new far address
		CA 08 00	RET 8	Stack	IP ← [SP + 1:SP]; SP ← SP + 2; CS ← [SP + 1:SP]; SP ← SP + 2 + 8	

[a]These two bytes are the 2's-complement offset between near memory location MEMN and the instruction immediately following the CALL instruction. For example, if the CALL is to be 12 locations forward, the offset is 00 0C. Similarly, 12 locations backward requires an offset of FFF4.

[b]MEMN is assumed to point to a near memory location (i.e., within the segment).

[c]MEMWDS is assumed to point to the word beginning at location 1000H in the data segment. The brackets ae optional because MEMWDS defines a word not a location to jump to.

[d]FAR PTR indicates that MEMF is located in a different code segment. In this case it is assumed that MEMF points at location 09D3H:1000H.

[e]MEMWWDS is assumed to point to the double word beginning at location 1000H in the data segment. The brackets are optional because MEMWWDS defines a double word, not a location to jump to.

[f]The same mnemonic is used for a near or far return. The assembler will generate the proper code based on the procedure definition statement (near or far).

Table 2.19 SOFTWARE INTERRUPT INSTRUCTIONS

General mnemonic		Object code	Coding example			
Op-code	Operand		Mnemonic	Segment for memory access	Symbolic operation	Description
INT	*type*	CD 23	INT 23H	Stack and interrupt jump table at 00000–003FFH	SP ← SP − 2; [SP + 1:SP] ← flags IF ← 0; TF ← 0; SP ← SP − 2; [SP + 1:SP] ← CS; CS ← [0008FH:0008EH];[a] SP ← SP − 2 [SP + 1:SP] ← IP; IP ← [0008DH:0008CH][b]	Save the flag, CS, and IP registers on the stack and transfer control to the far address stored in the double word beginning at absolute address *type* * 4
INTO	none	CE	INTO	Stack and interrupt jump table at 00000–003FFH	If OF = 1, then SP ← SP − 2 [SP + 1:SP] ← flags; IF ← 0; TF ← 0; SP ← SP − 2; [SP + 1:SP] ← CS; CS ← [00013H:00012H];[a] SP ← SP − 2; [SP + 1:SP] ← IP; IP ← [00011H;00010H][b]	If an overflow condition exists (OF = 1), a type 4 interrupt is executed
IRET	none	CF	IRET	Stack	IP ← [SP + 1:SP]; SP ← SP + 2; CS ← [SP + 1:SP]; SP ← SP + 2; flags ← [SP + 1:SP]; SP ← SP + 2	Transfer control back to the point of interrupt by popping the IP, CS, and flag registers from the stack; IRET is normally used to exit any interrupt procedure whether activated by hardware or software

[a][(type * 4) + 3: (type * 4) + 2].

[b][(type * 4) + 1: (type * 4)].

TABLE 2.20 PROCESSOR CONTROL INSTRUCTIONS

General mnemonic		Object code	Coding example			
Op-code	Operand		Mnemonic	Segment for memory access	Symbolic operation	Description
STC	None	F9	STC	Within CPU	CF ← 1	Set carry flag
CLC	None	F8	CLC	Within CPU	CF ← 0	Clear carry flag
CMC	None	F5	CMC	Within CPU	CF ← $\overline{CF}$	Complement carry flag
STD	None	FD	STD	Within CPU	DF ← 1	Set direction flag (auto decrement for string instructions)
CLD	None	FC	CLD	Within CPU	DF ← 0	Clear direction flag (auto increment for string instructions)
STI	None	FB	STI	Within CPU	IF ← 1	Set interrupt flag (enabling interrupts on the INTR line)
CLI	None	FA	CLI	Within CPU	IF ← 0	Clear interrupt flag (disabling interrupts on the INTR line)
HLT	None	F4	HLT	Within CPU	None	Halt
WAIT	None	9B	WAIT	Within CPU	None	Enter wait state if $\overline{TEST}$ line = 1
LOCK	*instruction*	F0 A1 00 10	LOCK MOV AX,MEMWDS[a]	Data	None	Output line $\overline{LOCK}$ = 0 while the instruction following LOCK executes; used to prevent coprocessors from accessing the bus during a particular instruction
NOP	None	90	NOP	Data	None	No operation
ESC	*number, source*	DE 0E 00 10	ESC 31H,MEMWDS[a]	Data	Data bus ← [MEMWDS]	Place the contents of the memory source operand on the data bus and execute a NOP; the first operand identifies a particular escape instruction to be executed by a coprocessor

[a]MEMWDS is assumed to point at the word beginning at location 1000H in the data segment.

8 Debug

In this chapter, you will learn

- How to start the **debug** utility
- How to use the **debug** commands and parameters

Note You need to read this chapter only if you are writing or testing executable programs (*.com* or *.exe* files) or object (*.obj*) files.

Introduction

The **debug** utility is a debugging program that provides a controlled testing environment for binary and executable object files. Note that **Edlin**, the MS-DOS line editor, is used to alter source files; **debug** is **Edlin**'s counterpart for binary files.

Debug eliminates the need to reassemble a program to see if a problem has been fixed by a minor change. It allows you to alter the contents of a file or the contents of a CPU register, and then immediately reexecute a program to check the validity of the changes made.

All **debug** commands may be aborted at any time by pressing CONTROL-C. The CONTROL-S key sequence suspends the display, so that you can read it before the output scrolls away. Pressing any key other than CONTROL-C or CONTROL-S restarts the display. All these commands are consistent with the control character functions available at the MS-DOS command level.

How to Start Debug

Starting Debug

Debug may be started two ways. By the first method, you type all commands in response to the **debug** prompt (a hyphen). By the second method, you type all commands on the line used to start **debug**.

Method 1: **debug**
Method 2: **debug** [*filename* [*arglist*]]

Method 1

Method 1: Debug

To start **debug** using method 1, simply type the following:

```
debug
```

Debug responds with the hyphen (-) prompt, signaling that it is ready to accept your commands. Since you didn't specify a filename, you can use other commands to work on current memory, disk sectors, or disk files.

Warnings

- When **debug** (version 3.0) is started, it sets up a program header at offset 0 in the program work area. In previous versions of **debug**, you could overwrite this header. You can still overwrite the default header if you don't give a *filename* to **debug**. If you are debugging a *.com* or *.exe* file, however, do not tamper with the program header below address 5CH, or **debug** will terminate.
- Do not restart a program after the following message is displayed:

  ```
  Program terminated normally
  ```

  You must reload the program with the N (name) and L (load) commands for it to run properly.

Copied pursuant to S53B of the Copyright Act as amended on behalf of Murdoch Uni. on 17.2.89

DEBUG COMMANDS H1
Extract from: Microsoft MS-DOS User's Guide and User's Reference p223-256, Microsoft Corporation 1987 pp 386. ISBN - not given in original.

Method 2: Command Line

To start **debug** using a command line, you must use the following syntax:

debug [*filename* [*arglist*]]

For example, if you specify a *filename*, the following would be a typical command to start **debug**:

```
debug file.exe
```

Debug would then load *file.exe* into memory starting at 100 (hexadecimal) in the lowest available segment. The **BX:CX** registers are loaded with the number of bytes placed into memory.

If you do include a *filename*, you might also specify an *arglist*. An *arglist* is a list of filename parameters and switches that are to be passed to the program *filename*. So when *filename* is loaded into memory, it is loaded as if it had been started with a command of the form, **debug** *filename arglist*.

Here, *filename* is the file to be debugged, and *arglist* is the rest of the command line used when **debug** calls and loads *filename* into memory.

Debug Command Information

Each **debug** command consists of a single letter followed by one or more parameters. Additionally, the control characters and special editing functions described in Chapter 5, "MS-DOS Editing and Function Keys," apply to **debug** as well.

If a syntax error occurs in a **debug** command, **debug** reprints the command line and indicates the error with a caret (^) and the word "Error" as in the following example,

```
dcs:100 cs:110
   ^ Error
```

Note that when typing commands and parameters you may use any combination of uppercase and lowercase letters.

The **debug** commands are listed. Following this list, the commands and their parameters are described in greater detail.

Method 2

About Debug commands

How to use the Debug command parameters

Debug command	Function
A [*address*]	Assemble
C *range address*	Compare
D [*range*]	Dump
E *address* [*list*]	Enter
F *range list*	Fill
G [=*address* [*address...*]]	Go
H *value value*	Hex
I *value*	Input
L [*address* [*drive:record record*]]	Load
M *range address*	Move
N *filename* [*filename*]	Name
O *value byte*	Output
Q	Quit
R [*register-name*]	Register
S *range list*	Search
T [=*address*] [*value*]	Trace
U [*range*]	Unassemble
W [*address* [*drive:record record*]]	Write

Debug Command Parameters

All **debug** commands accept parameters, except the Q (quit) command. Parameters may be separated by delimiters (spaces or commas), but a delimiter is required only between two consecutive hexadecimal values. Thus, the following commands are equivalent:

```
dcs:100 110
d cs:100 110
d,cs:100,110
```

Parameter	Definition
drive:	A one-digit hexadecimal value that indicates which drive a file will be loaded from or written to. The valid values are 0–3, where 0 = A:, 1 = B:, 2 = C:, 3 = D:.
byte	A two-digit hexadecimal value placed in or read from an address or register.
record	One-digit to three-digit hexadecimal value that indicates the logical record number on the disk and the number of disk sectors to be written or loaded. Logical records correspond to sectors; however, since they represent the entire disk space, their numbering differs.
value	A hexadecimal value of up to four digits that specifies a port number or the number of times a command should repeat its functions.

address	A two-part designation containing either an alphabetic segment register or a four-digit segment address plus an offset value. You may omit the segment name or segment address, in which case the default segment DS is used for all commands except G, L, T, U, and W, for which the default segment is CS. All numeric values are hexadecimal.
	Following is an example *address*:
	`CS:0100`
	`04BA:0100`
	Note that the colon is required between the segment name (whether numeric or alphabetic) and the offset value.

range	Contains two addresses: for example, *address address*; or one address, an L, and a value: for example, *address* L *value* where *value* is the number of lines on which the command should operate (L80 is assumed). The second type of *range* cannot be used if another hexadecimal value follows, since the hexadecimal value would be interpreted as the second *address* of the *range*.
	Here are some example ranges:
	`CS:100 110`
	`CS:100 L 10`
	`CS:100`
	The following example, however, is illegal:
	`CS:100 CS:110`
	`b Error`
	The limit for *range* is 10000 (hexadecimal). To specify a *value* of 10000 with only four digits, type *0000* (or *0*).

list	A series of *byte* values or *strings*. *List* must be the last parameter on the command line. Following is an example *list*: `fcs:100 42 45 52 54 41`

string	Any number of characters enclosed in quotation marks. The quotation marks may be either single (' ') or double (" "). If the delimiter marks must appear within a *string*, you must use the double quotation marks.
	For example, the following strings are legal:
	`"This 'string' is okay."`
	`"This ""string"" is okay."`
	However, this string is illegal:
	`""This "string" is not okay.""`
	Note that the double quotation marks are not necessary in the following strings:
	`"This ''string'' is not necessary."`
	`'This ""string"" is not necessary.'`
	The ASCII values of the characters in the string are used as a *list* of byte values.

Assemble

(A)ssemble

Purpose:

Assembles 8086/8087/8088 mnemonics directly into memory.

Syntax:

A[*address*]

Comments:

If it finds a syntax error, **debug** responds with the following message, then redisplays the current assembly address:

```
^Error
```

All numeric values are hexadecimal and you must type them as 1–4 characters. Also, you must specify prefix mnemonics *in front of* the opcode to which they refer. You may type them on a separate line, however.

The segment override mnemonics are **CS:**, **DS:**, **ES:**, and **SS:**. The mnemonic for the far return is **RETF**. String manipulation mnemonics must explicitly state the string size. For example, use **MOVSW** to move word strings, and use **MOVSB** to move byte strings.

The assembler will automatically assemble short, near, or far jumps and calls, depending on byte displacement, to the destination address. You may override these jumps and calls by using a **NEAR** or **FAR** prefix, as in the following example:

```
0100:0500 JMP   502        ; a 2-byte short jump
0100:0502 JMP   NEAR 505   ; a 3-byte near jump
0100:505  JMP   FAR  50A   ; a 5-byte far jump
```

You may abbreviate the **NEAR** prefix to **NE**, but the **FAR** prefix cannot be abbreviated.

Debug cannot tell whether some operands refer to a word memory location or to a byte memory location. In this case, the data type must be explicitly stated with the prefix, **WORD PTR** or **BYTE PTR**. Acceptable abbreviations are **WO** and **BY**. For example,

```
NEG     BYTE PTR [128]
DEC     WO [SI]
```

Debug also cannot tell whether an operand refers to a memory location or to an immediate operand, so it uses the common convention that operands enclosed in square brackets refer to memory locations. For example,

```
MOV     AX,21    ; Load AX with 21H
MOV     AX,[21]  ; Load AX with the
                 ; contents
                 ; of memory location 21H
```

Two popular pseudo-instructions are available with the A (assemble) command: the **DB** opcode, which assembles byte values directly into memory; and the **DW** opcode, which assembles word values directly into memory. Following are examples of both:

```
DB      1,2,3,4,"THIS IS AN EXAMPLE"
DB      'THIS IS A QUOTATION MARK: "'
DB      "THIS IS A QUOTATION MARK: '"

DW      1000,2000,3000,"BACH"
```

The **A** command supports all forms of register indirect commands. For example,

```
ADD     BX,34[BP+2].[SI-1]
POP     [BP+DI]
PUSH    [SI]
```

All opcode synonyms are also supported, as in the next example:

```
LOOPZ   100
LOOPE   100

JA      200
JNBE    200
```

For 8087 opcodes, the **WAIT** or **FWAIT** prefixes must be explicitly specified, as in these last examples:

```
FWAIT FADD ST,ST(3)      ; This line assembles
                         ; an FWAIT prefix
LD TBYTE PTR [BX]        ; This line does not
```

Compare

Purpose:

Compares the portion of memory specified by *range* to a portion of the same size beginning at the specified *address*.

Syntax:

Crange address

Comments:

If the two areas of memory are identical, there is no display, and **debug** returns with the MS-DOS prompt. If there *are* differences, they are displayed in this format:

address1 byte1 byte2 address2

Example:

The following commands have the same effect:

```
C100,1FF 300
```

or:

```
C100L100 300
```

Each command compares the block of memory from 100 to 1FFH with the block of memory from 300 to 3FFH.

(C)ompare

Comparing blocks of memory

(D)ump

Dump

Purpose:

Displays the contents of the specified *range* of memory.

Syntax:

D[*range*]

Comments:

If you specify a *range* of addresses with the **D** (dump) command, the contents of the *range* are displayed. If you don't use parameters with the **D** command, 128 bytes are displayed at the first address (DS:100) after the address displayed by the previous **D** command.

The dump is displayed in two portions: a hexadecimal dump (each byte is shown in hexadecimal value) and an ASCII dump (the bytes are shown in ASCII characters). Nonprinting characters are denoted by a period (.) in the ASCII portion of the display. Each display line shows 16 bytes, with a hyphen between the eighth and ninth bytes. Each displayed line begins on a 16-byte boundary. At times in this chapter, displays are split to fit them on the page.

Examples:

If you type the command

```
dcs:100 110
```

debug displays the dump in the following format:

```
04BA:0100 42 45 52 54 41 ... 4E 44 TOM SAWYER
```

If you simply type the **D** command, the display is formatted as just described. Each line of the display begins with an address incremented by 16 from the address on the previous line.

Each subsequent **D** (typed without parameters) displays the bytes immediately following those last displayed.

Using the Dump command

If you type the following command, the display is formatted as described above, but 20H bytes are displayed:

```
DCS:100 L 20
```

If you then type the following command, the display is formatted as described above, but all the bytes in the range of lines from 100H to 115H in the CS segment are displayed:

```
DCS:100 115
```


(E)nter

Enter

Purpose:

Enters byte values into memory at the specified *address*.

Syntax:

E*address*[*list*]

Comments:

If, when using the **E** (enter) command, you type the optional *list* of values, the byte values are replaced automatically. (If an error occurs, no byte values are changed.)

If you type the *address* without the optional *list*, **debug** displays the *address* and its contents, repeats the *address* on the next line, and then waits for your input. At this point, the E command waits for you to perform one of the following actions:

- Replace a byte value with a value you type. Simply type the value after the current value. If the one you type is not a legal hexadecimal value or if it contains more than two digits, the illegal or extra character is not echoed.
- Press the SPACEBAR to advance to the next byte. To change the value, simply type the new value as described in the previous action. If, when you press the SPACEBAR, you move beyond an 8-byte boundary, **debug** starts a new display line with the address displayed at the beginning.
- Type a hyphen (-) to return to the preceding byte. If you decide to change a byte behind the current position, typing the hyphen returns the current position to the previous byte. When you type the hyphen, a new line is started with its address and byte value displayed.
- Press the RETURN key to terminate the **E** command. The RETURN key may be pressed at any byte position.

Examples:

Suppose you type the following command:

```
ECS:100
```

Now suppose that **debug** displays the following:

```
04BA:0100 EB._
```

To change this value to, say, 41, type the number 41 at the cursor, as shown:

```
04BA:0100 EB.41_
```

To step through the subsequent bytes, you would press the SPACEBAR until you saw the following:

```
04BA:0100 EB.41  10. 00.  BC._
```

To change BC to the number 42, for instance, you would type the number at the cursor, as follows:

```
04BA:0100 EB.41  10. 00.  BC.42_
```

Notice that the value 10 should be 6F. To correct this value, you would type the hyphen as many times as needed to return to byte 0101 (value 10), then replace 10 with 6F:

```
04BA:0100  EB.41    10.  00.   BC.42-
04BA:0102  00.-_
04BA:0101  10.6F_
```

Pressing the RETURN key ends the **E** command and returns you to the **debug** command level.

Using the Enter command

(F)ill

Fill

Purpose:

Fills the addresses in the specified *range* with the values in the specified *list*.

Syntax:

F*range list*

Comments:

If the *range* contains more bytes than the number of values in the *list*, the *list* will be used repeatedly until all bytes in the *range* are filled.

If the *list* contains more values than the number of bytes in the *range*, the extra values in the *list* are ignored. If any of the memory in the *range* is not valid (bad or nonexistent), the error will occur in all succeeding locations.

Example:

Suppose you type the following command:

```
F04BA:100 L 100 42 45 52 54 41
```

In response, **debug** would fill memory locations 04BA:100 through 04BA:1FF with the bytes specified. The five values would then be repeated until all the 100H bytes were filled.

Using the Fill command

Go

Purpose:

Executes the program currently in memory.

Syntax:

G[=*address* [*addresses*]]

Comments:

If you type the G (go) command by itself, the program currently in memory executes as if it had run outside **debug**.

If you set =*address*, execution of the **G** command begins at the address specified. The equal sign (=) is required so that **debug** can distinguish the start =*address* from the breakpoint *addresses*.

With the other optional *addresses* set, execution stops at the first *address* encountered, regardless of that address' position in the list of addresses that halt execution or program branching. When program execution reaches a breakpoint, the registers, flags, and decoded instruction are displayed for the last instruction executed. The result is the same as if you had typed the **R** (register) command for the breakpoint address.

You may set up to ten breakpoints, but only at addresses containing the first byte of an 8086 opcode. If you set more than ten breakpoints, **debug** returns the **BP** error message.

The user stack pointer must be valid and must have 6 bytes available for this command. The **G** command uses an **IRET** instruction to cause a jump to the program under test. The user stack pointer is set, and the user flags, code segment register, and instruction pointer are pushed on the user stack. (If the user stack is not valid or is too small, the operating system may crash.) An interrupt code (0CCH) is placed at the specified breakpoint address(es).

When **debug** encounters an instruction with the breakpoint code, it restores all breakpoint addresses to their original instructions. If you don't halt execution at one of the breakpoints, the interrupt codes are not replaced with the original instructions.

Using the Go command

Example:

Suppose you type the following command:

```
GCS:7550
```

The program currently in memory would execute up to the address 7550 in the CS segment. **Debug** would then display registers and flags, after which the **G** command would terminate.

After **debug** has encountered a breakpoint, if you type the **G** command again the program runs as if you had typed the filename at the MS-DOS command level. The only difference is that program execution begins at the instruction after the breakpoint, rather than at the usual start address.

Hex

Purpose:

Performs hexadecimal arithmetic on the two specified parameters.

Syntax:

H*value value*

Comments:

First, **debug** adds the two parameters, then subtracts the second parameter from the first. The results of these actions are displayed on one line — first the sum, then the difference.

Example:

Suppose you type the following command:

```
H19F 10A
```

In response, **debug** would perform the calculations and then display the following result:

```
02A9 0095
```


(H)ex

Using the Hex command

(I)nput

Using the Input command

Input

Purpose:

Inputs and displays one byte from the port specified by *value*.

Syntax:

I*value*

Comment:

A 16-bit port address is allowed.

Example:

Suppose you type the following command:

```
I2F8
```

Suppose also that the byte at the port is 42H. **Debug** would input the byte and then display the following:

```
42
```

Load

Purpose:

Loads a file into memory.

Syntax:

L[*address* [*drive*: *record record*]]

Comments:

Set BX:CX to the number of bytes read. The file must have been named either when you started **debug** or with the N (name) command. Both the **debug** invocation and the N command format a filename properly in the normal format of a file control block at CS:5C.

If you use the L (load) command without any parameters, **debug** loads the file into memory beginning at address CS:100 and sets **BX:CX** to the number of bytes loaded. If you type the L command with an address parameter, loading begins at the memory location specified by the *address*. If you use the L command with all parameters included, absolute disk sectors are loaded, instead of a file.

Each *record* is taken from the specified *drive*: (the drive name is numeric: 0 = A:, 1 = B:, 2 = C:, etc.). **Debug** begins loading with the first specified *record*, and continues until the number of sectors in the second *record* have been loaded.

Example:

Suppose, once you have started **debug**, that you type the following commands:

```
-NFILE.COM
```

Now, to load *file.com*, you would simply type the L command.

Debug would then load the file and display the **debug** prompt. Suppose now that you want to load only portions of a file or certain records from a disk. To do this, you would type the following:

```
L04BA:100 2 0F 6D
```

Debug would then load 109 (6DH) records, beginning with logical record number 15, into memory beginning at address 04BA:0100. Then once it had loaded the records, **debug** would simply return the hyphen (-) prompt.

(L)oad

Using the Load command

If the file has an *.exe* extension, it would be relocated to the load address specified in the header of the *.exe* file. The *address* parameter is always ignored for *.exe* files. The header itself is stripped off the *.exe* file before it is loaded into memory. So, the size of an *.exe* file on disk will differ from its size in memory.

If the file was named by the N (name) command, or specified when you started **debug**, as a *.hex* file, then typing the L command with no parameters would cause **debug** to load the file beginning at the address specified in the *.hex* file. If the L command included the option *address*, **debug** would add the *address* specified in the L command to the address found in the *.hex* file to determine the start address at which to load the file.

Move

Purpose:

Moves the block of memory specified by *range* to the location beginning at the specified *address*.

Syntax:

Mrange address

Comments:

Overlapping moves (i.e., moves where part of the block overlaps some of the current addresses) are always performed without loss of data. Addresses that could be overwritten are moved first. For moves from higher to lower addresses, the sequence of events is to first move the data beginning at the block's lowest address and then work toward the highest. For moves from lower to higher addresses, the sequence is to first move the data beginning at the block's highest address and then work toward the lowest.

Note that if the addresses in the block being moved will not have new data written to them, the data in the block *before the move* will remain. The **M** (move) command copies the data from one area into another, in the sequence described, and writes over the new addresses. This action is why the sequence of the move is important.

Example:

Suppose you type the following command:

```
MCS:100 110 CS:500
```

In response, **debug** would first move address CS:110 to CS:510, then CS:10F to CS:50F, and so on until it has moved CS:100 to CS:500. To review the results of the move, you could type the **D** (dump) command, using the same *address* you used with the **M** command.

Using the Move command

Name

Purpose:

Sets filenames.

Syntax:

N*filename* [*filename...*]

Comments:

The **N** (name) command performs two functions. First, it assigns a filename for a later **L** (load) or **W** (write) command. So if you start **debug** without naming a file to be debugged, you must type the command N*filename* before a file can be loaded. Second, the **N** command assigns filename parameters to the file being debugged. In this case, **N** accepts a list of parameters used by the file being debugged.

Note that these two functions overlap. Consider, for example, the following set of **debug** commands:

```
-NFILE1.EXE
-L
-G
```

The **N** command would use these commands to perform the following steps:

1. It would first assign the filename *file1.exe* to the file to be used in any later **L** or **W** commands.
2. It would also assign the *file1.exe* filename to the first filename parameter used by any program that is later debugged.
3. The **L** command would then load *file1.exe* into memory.
4. The **G** (go) command would cause *file1.exe* to be run with *file1.exe* as the single filename parameter (that is, *file1.exe* would be run as if *file1.exe* had been typed at the command level).

A more useful chain of commands might look like this:

```
-NFILE1.EXE
-L
-NFILE2.DAT FILE3.DAT
-G
```

In this example, the N command sets *file1.exe* as the filename for the subsequent L command, which loads *file1.exe* into memory. The N command is then used again, this time to specify the parameters to be used by *file1.exe*. Finally, when the G command is run, *file1.exe* is executed as if *file1 file2.dat file3.dat* had been typed at the MS-DOS command level.

Note that if you were to execute a W command now, then *file1.exe* — the file being debugged — would be saved with the name *file2.dat*. To avoid this kind of result, you should always execute an N command before either an L or W command.

There are four regions of memory that can be affected by the N command:

Memory location	Contents
CS:5C	FCB for file 1
CS:6C	FCB for file 2
CS:80	Count of characters
CS:81	All characters typed

The first filename parameter that you specify for the N command has a file control block (FCB) set up at CS:5C. If you name a second filename parameter, an FCB is set up for this parameter beginning at CS:6C. The number of characters typed in the N command (exclusive of the first character, N) is given at location CS:80.

The actual stream of characters given by the N command (again, exclusive of the letter N) begins at CS:81. Note that this stream of characters may contain switches and delimiters that would be legal in any command typed at the MS-DOS command level.

Example:

A typical use of the N command is as follows:

```
 DEBUG PROG.COM
-NPARAM1 PARAM2/C
-G
-
```

In this case, the G command executes the file in memory as if you had typed the following command line:

```
PROG PARAM1 PARAM2/C
```

Testing and debugging therefore reflect a normal run-time environment for *prog.com*.

Using the Name command

(O)utput

Output

Purpose:

Sends the specified *byte* to the output port specified by *value*.

Syntax:

O*value byte*

Comment:

A 16-bit port address is allowed.

Example:

Using the Output command

Suppose you want to **debug** to output the byte value 4F to output port 2F8. To do this, you could simply type the following command:

```
02F8 4F
```

Quit

Purpose:

Terminates the **debug** utility.

Syntax:

Q

Comments:

The Q (quit) command takes no parameters and exits **debug** without saving the file you're currently working with. You are returned to the MS-DOS command level.

Example:

To end the debugging session, simply type the following and press the RETURN key:

```
q
```

Debug terminates, and control returns to the MS-DOS command level.

(Q)uit

Using the Quit command

(R)egister

Register

Purpose:

Displays the contents of one or more CPU registers.

Syntax:

R[*register-name*]

Comments:

If you do not type a *register-name*, the **R** (register) command dumps the register storage area and displays the contents of all registers and flags.

If you do type a *register-name*, the 16-bit value of that register is displayed in hexadecimal, and a colon then appears as a prompt. You can now either type a *value* to change the register, or press the RETURN key if you don't want a change.

Following is a list of the valid *register-names*:

AX	**BP**	**SS**	
BX	**SI**	**CS**	
CX	**DI**	**IP**	(**IP** and PC both refer
DX	**DS**	**PC**	to the instruction pointer.)
SP	**ES**	**F**	

Any other entry for *register-name* results in a **BR** error message.

If you type **F** as the *register-name*, **debug** displays each flag with a two-character alphabetic code. To change any flag, type the opposite two-letter code. The flags are then either set or cleared.

The flags are listed below with their codes for **SET** and **CLEAR**:

Flag name	Set	Clear
Overflow	**OV**	**NV**
Direction	**DN** (Decrement)	**UP** (Increment)
Interrupt	**EI** (Enabled)	**DI** (Disabled)
Sign	**NG** (Negative)	**PL** (Plus)
Zero	**ZR**	**NZ**
Auxiliary Carry	**AC**	**NA**
Parity	**PE** (Even)	**PO** (Odd)
Carry	**CY**	**NC**

Whenever you type the **RF** command, the flags are displayed (in the order shown in the previous table) in a row at the beginning of a line. At the end of the list of flags, **debug** displays a hyphen (-).

You may enter new flag values in any order as alphabetic pairs. You do not have to leave spaces between these values. To exit the **R** command, press the RETURN key. Any flags for which you did not specify new values remain unchanged.

If you type more than one value for a flag, **debug** returns a **DF** error message. If you enter a flag code other than one of those shown in the table above, **debug** returns a **BF** error message. In both cases, the flags up to the error in the list are changed; those flags at and after the error are not.

When you start **debug**, the segment registers are set to the bottom of free memory, the instruction pointer is set to 0100H, all flags are cleared, and the remaining registers are set to zero.

Using the Register command

Examples:

If you type the following command, **debug** displays all registers, flags, and the decoded instruction for the current location:

```
R
```

If the location is CS:11A, for example, the display will look similar to this:

```
AX=0E00 BX=00FF CX=0007 DX=01FF SP=039D BP=0000
SI=005C DI=0000 DS=04BA ES=04BA SS=04BA CS=04BA
IP=011A  NV UP DI NG NZ AC PE NC
04BA:011A  CD21           INT     21
```

If you then type the following command, **debug** will display these flags:

```
RF
NV UP DI NG NZ AC PE NC - _
```

Now, you could type any valid flag designation, in any order, with or without spaces. For example,

```
NV UP DI NG NZ AC PE NC - PLEICY
```

In response, **debug** would display the **debug** prompt. To see the changes, type either the **R** or **RF** command:

```
RF
NV UP EI PL NZ AC PE CY - _
```

Press the RETURN key to leave the flags this way or to specify different flag values.

Search

Purpose:

Searches the specified *range* for the specified *list* of bytes.

Syntax:

Srange list

Comments:

The *list* may contain one or more bytes, each separated by a space or comma. If the *list* contains more than one byte, only the first address of the byte string is returned. If the *list* contains only one byte, all addresses of the byte in the *range* are displayed.

Example:

Suppose you type the following command:

```
SCS:100 110 41
```

Debug would display a response similar to

```
04BA:0104
04BA:010D
-type:
```

(S)earch

Using the Search command

(T)race

Trace

Purpose:

Executes one instruction and displays the contents of all registers, flags, and the decoded instruction.

Syntax:

T[=*address*] [*value*]

Comments:

If you include the =*address* option in the T (trace) command, tracing occurs at the specified =*address*. The *value* option causes **debug** to execute and trace the number of steps specified by *value*.

The T command uses the hardware trace mode of the 8086 or 8088 microprocessor. Consequently, you may also trace instructions stored in ROM (Read Only Memory).

Example:

Suppose you type the following command:

```
T
```

In response, **debug** would return a display of the registers, flags, and decoded instruction for that one instruction. Assuming, for this example, that the current position is 04BA:011A, **debug** might return the following display:

```
AX=0E00 BX=00FF CX=0007 DX=01FF SP=039D BP=0000
SI=005C DI=0000 DS=04BA ES=04BA SS=04BA CS=04BA
IP=011A  NV UP DI NG NZ AC PE NC
04BA:011A  CD21           INT      21
```

If you type the following command, **debug** executes sixteen (10 hex) instructions beginning at 011A in the current segment, and then displays all registers and flags for each instruction as it is executed. The display scrolls away until the last instruction is executed, and then stops. Now you can see the register and flag values for the last few instructions performed:

```
T=011A 10
```

Remember that if you want to study the registers and flags for any instruction (at any time), you can press CONTROL-S to stop the display from scrolling.

Using the Trace command

Unassemble

Purpose:

Disassembles bytes and displays the source statements that correspond to them, with addresses and byte values.

Syntax:

U[*range*]

Comments:

The display of disassembled code looks like a listing for an assembled file. If you type the U (unassemble) command without parameters, 20 hexadecimal bytes are disassembled at the first address after that displayed by the previous U command. If you type the U command and include the *range* parameter, **debug** disassembles all bytes in *range*. But if you specify *range* only as an *address*, 20H bytes are disassembled.

Example:

Suppose you type the following command:

```
U04BA:100 L10
```

In response, **debug** would disassemble 16 bytes, beginning at address 04BA:0100:

```
04BA:0100  206472      AND   [SI+72],AH
04BA:0103  69          DB    69
04BA:0104  7665        JBE   016B
04BA:0106  207370      AND   [BP+DI+70],DH
04BA:0109  65          DB    65
04BA:010A  63          DB    63
04BA:010B  69          DB    69
04BA:010C  66          DB    66
04BA:010D  69          DB    69
04BA:010E  63          DB    63
04BA:010F  61          DB    61
```


Using the Unassemble command

Now, suppose you type the following:

```
U04BA:0100 0108
```

The display would now show:

```
04BA:0100  206472      AND   [SI+72],AH
04BA:0103  69          DB    69
04BA:0104  7665        JBE   016B
04BA:0106  207370      AND   [BP+DI+70],DH
```

If the bytes in some addresses are altered, the disassembler alters the instruction statements. You can then type the U command for the changed locations, for the new instructions viewed, and for the disassembled code used to edit the source file.

TABLE 2.13 ARITHMETIC ADJUST INSTRUCTIONS

General mnemonic		Object code	Coding example			
Op-code	Operand		Mnemonic	Segment for memory access	Symbolic operation	Description
DAA	none	27	DAA	Within CPU	If AL · 0F > 9 or AF = 1, then AL ← AL + 6; AF ← 1 If AL > 9F or CF = 1, then AL ← AL + 60H; CF ← 1	Adjust AL to a pair of valid packed decimal digits following the addition of two valid packed or unpacked decimal operands; all flags except OF (undefined) are affected
DAS	none	2F	DAS	Within CPU	If AL · 0F > 9 or AF = 1, then AL ← AL − 6; AF ← 1 If AL > 9F or CF = 1, then AL ← AL − 60H; CF ← 1	Adjust AL to a pair of valid packed decimal digits following the subtraction of two valid packed or unpacked decimal operands; all flags except OF (undefined) are affected
AAA	none	37	AAA	Within CPU	If AL · 0F > 9 or AF = 1, then AL ← AL + 6; AH ← AH + 1; AF ← 1; CF ← AF; AL ← AL · 0F	Adjust AL to a single unpacked decimal number following the addition of two valid unpacked decimal operands; the high-order half-byte of AL is zeroed and AH is incremented by 1; all flags except AF and CF are undefined

00 0E AL = AL+6
AL = 14
61 0E AH = AH+1 = 01

TABLE 2.13 (continued)

General mnemonic		Object code	Coding example			
Op-code	Operand		Mnemonic	Segment for memory access	Symbolic operation	Description
AAS	none	3F	AAS	Within CPU	If AL · 0F > 9 or AF = 1, then AL ← AL − 6; AH ← AH − 1; AF ← 1; CF ← AF; AL ← AL · 0F	Adjust AL to a single unpacked decimal number following the subtraction of two valid unpacked decimal operands; the high-order half-byte of AL is zeroed and AH is decremented by 1; all flags except AF and CF are undefined
AAM	none	D4 0A	AAM	Within CPU	AH ← AL/0AH AL ← remainder	Following the multiplication of two valid unpacked decimal operands, AAM converts the result in AL to two valid unpacked decimal digits in AH and AL; all flags except PF, SF, and ZF are undefined

AAD	none	D5 0A	AAD	Within CPU	AL ← (AH * 0AH) + AL AH ← 0	Before dividing AX by a single-digit unpacked decimal operand, AAD converts the two-digit unpacked decimal number in AX to a binary number in AL and 0 in AH; the quotient produced by the following division will then be a valid unpacked decimal number in AL and remainder in AH; all flags except PF, SF, and ZF are undefined
CBW	none	98	CBW	Within CPU	If AL < 80H, then AH ← 0 If AL > 7F, then AH ← FFH	Before dividing AX by a byte operand, CBW extends the sign of a byte dividend in AL into AH, thus converting AL into a valid signed word in AX; none of the flags are affected
CWD	none	99	CWD	Within CPU	If AX < 8000H, then DX ← 0 If AX > 7FFFH, then DX ← FFFFH	Same as CBW but extends the sign of a word dividend in AX into a double word in DX:AX; none of the flags are affected

The result in AX is now encoded in ASCII form and represents the correct decimal result: 12.

When multiplying two unpacked decimal numbers the result is almost always an invalid decimal number. For example, if AL = 06H and CL = 09H, the instruction MUL CL leaves 36H (54D) in AL. The correct binary result but incorrect decimal result. The AAM (ASCII adjust for multiplication) instruction converts this result to two valid unpacked decimal numbers in AH and AL. These digits can then be easily converted to ASCII if desired.

Example 2.16

Write a program to multiply the two ASCII digits in CH and CL and leave the ASCII result in AH and AL. Determine the contents of AX if CX = 3639H.

Solution The program is as follows:

```
AND   CX,OFOFH      ;Unpack CH and CL—CX = 0609H
MOV   AL,CH         ;Multiplier to AL
MUL   CL            ;Form product—AX = 0036H
AAM                 ;Form two unpacked decimal digits
                    ;AX = 0504H
ADD   AX,3030H      ;Add the ASCII bias—AX = 3534H
```

It might also be worth noting that the AAM instruction can be considered a special form of the DIV instruction. This is because it divides AL by 10, leaving the quotient in AH and the remainder in AL.

When dividing a two-digit unpacked decimal number in AX by an unpacked decimal operand the AAD (ASCII adjust for division) instruction should be used *before* the division. This will convert AX to a binary number in AL and 0 in AH. (Remember: The largest decimal number in AX would be 99, which can be represented as the single binary byte 63H—thus AH is made 0.) The result of the division will then be two unpacked decimal numbers—the quotient in AL and the remainder in AH.

As an example, if AX = 0607H (67D) and CL = 09H, AX can be divided by CL using the following sequence.

```
AAD           ;Convert AX to 0043H (67D)
DIV   CL      ;AL = 07 and AH = 04 (7 remainder 4)
```

When signed division is performed, the CBW and CWD instructions may be required. For example, if it is desired to divide two signed bytes, a problem occurs. IDIV requires one of the digits to be in AX—a 16-bit register. CBW solves the problem by converting the byte in AL to a word in AX. It does this by extending the sign of the byte in AL to AH. Thus the number FBH (−5) becomes FFFBH in AX following the CBW instruction.

CWD performs the same function when it is desired to divide two signed 16-bit words. Following the CWD instruction, the signed word in AX is converted to a signed *double word* in DX:AX.

SELF-REVIEW 2.7 (answers on page 108)

2.7.1. Determine the contents of register AL after the following instructions are executed.

```
MOV  AL,3AH
MOV  CH,0A9H
ADD  CH,06H
ADD  AL,CH
NEG  AL
DEC  AL
```

2.7.2. Determine the contents of register AX after the following instructions are executed.

```
MOV  AX,6242H
MOV  BL,0FAH
DIV  BL
```

2.7.3. C3H is a valid packed number. (True/False)

2.7.4. When decimal addition is to be performed, the DAA instruction should be used immediately _____ the addition instruction.

2.7.5. If, following an addition, register AX = 000EH, executing the AAA instruction will cause register AX to become _____.

2.7.6. Determine the final contents of register AX in Ex. 2.16 if CX = 3336H initially.

2.7.7. If AX = 0805H and CL = 09H, determine the final contents of register AX after the following two instructions are executed.

```
AAD
DIV  CL
```

2.7.8. You would use the CWD instruction before dividing two signed words, forcing the dividend to become a signed double word. (True/False)

2.8 TRANSFER OF CONTROL INSTRUCTIONS

The stored program computer repeatedly follows the sequence: fetch the instruction whose address is in IP, increment IP, and execute the instruction. This implies that all programs will execute in a sequential manner. However, there are times when it is necessary to transfer program control to an address that is not the next instruction in sequence.

Examples are groups of instructions that must be executed repeatedly, groups of instructions that are shared throughout a program (subroutines), conditional transfers based on the state of the flags, and software interrupts. In this section we look at the 8086/88 instructions that allow program control to be transferred to new (nonsequential) addresses.

Unconditional Jump Instructions

Table 2.14 lists the five forms of the jump instruction. The first three are *unconditional*, meaning that control is transferred to the target address without regard for the state of

TABLE 2.14 JUMP INSTRUCTIONS

General mnemonic		Object code	Coding example			
Op-code	Operand		Mnemonic	Segment for memory access	Symbolic operation	Description
JMP	*near target*	E9 — — [a]	JMP MEMN[b]	Code	IP ← MEMN	Transfer control to (near) target location within the segment; the addressing mode may be direct, memory indirect, or register indirect
		FF 26 00 10	JMP [MEMWDS][c]	Data	IP ← [MEMWDS + 1:MEMWDS]	
		FF 27	JMP [BX]	Data	IP ← [BX + 1:BX]	
		FF E0	JMP AX	Within CPU	IP ← AX	
JMP	SHORT *target*	EB — [d]	JMP SHORT MEMS[e]	Code	IP ← MEMS	Transfer control to the (short) target location; there are no indirect forms of this instruction
JMP	*far target*	EA 03 00 D3 9E	JMP FAR PTR MEMF[f]	Code	IP ← 0003H; CS ← 9ED3H	Transfer control to the far target location outside the segment
		FF 2E 05 10	JMP [MEMWWDS][g]	Data	IP ← [1006H:1005H]; CS ← [1008H:1007H]	
		FF 2F	JMP DWORD PTR[BX]	Data	IP ← [BX + 1:BX]: CS ← [BX + 3:BX + 2]	

Jcond[h] *short target*	73 — [d]	JNC MEMS	Code	If CF = 0, then IP ← MEMS	Transfer control to the short target address if the condition is true; all conditional jumps require short targets
JCXZ *short target*	E3 — [d]	JCXZ MEMS	Code	If CX = 0, then IP ← MEMS	Transfer control to the short target address if CX = 0

[a]These two bytes are the 2's-complement offset between near memory location MEMN and the instruction immediately following the JMP instruction. For example, if the jump is to be 12 locations forward, the offset is 00 0C. Similarly, 12 locations backward requires an offset of FFF4. The offset is limited to 32767 locations forward (7FFFH) and −32768 locations backward (8000H).

[b]MEMN is assumed to point to a near-memory location (i.e., within the segment).

[c]MEMWDS is assumed to point at the word beginning at location 1000H in the data segment. The brackets are optional because MEMWDS defines a word, not a location to jump to.

[d]This byte is the 2's-complement offset between short memory location MEMS and the instruction immediately following the jump SHORT instruction. The offset is limited to 127 locations forward (7FH) and 128 locations backward (80H).

[e]MEMS is assumed to point to a short memory location (i.e., +127 or −128 locations from the address of the instruction immediately following the JMP SHORT instruction). The SHORT operator is optional if MEMS is already known by the assembler.

[f]MEMF is assumed to point to the far memory location 9ED3H:0003H.

[g]MEMWWDS is assumed to point to the double word beginning at location 1005H in the data segment. The brackets are optional because MEMWWDS defines a double word, not a location to jump to.

[h]See Table 2.15 for a list of testable conditions.

the processor flags. This form is used when it is required to repeat a group of instructions For example, consider the program loop in Fig. 2.15.

This program repeatedly divides the data input at ports 6 and 7 by 4, outputting the result to port 9AH. The JMP REPEAT instruction transfers control back to the input instruction. REPEAT is a label given to the memory location storing the first byte of the IN AX,06H instruction. Note the column labeled *ADDR* in Fig. 2.15. This column keeps track of the logical address of each byte in the program. In this case you can see that REPEAT represents address 0002H and E5 is stored in this location (the operand 06H is stored in 0003H).

The direct form of the near jump (shown in Fig. 2.15) is also called a *relative jump*. This is because program control is transferred to a new address relative to the value in IP. The two bytes following the JMP op-code E9 are added to IP to form the target address.

Referring to Fig. 2.15, the assembler has generated the two bytes F7 FF for the target address REPEAT. Yet the IN AX,06H instruction is actually stored beginning at address 0002H. Does this make sense? The answer, of course, is yes if we do our arithmetic correctly. After fetching the jump instruction IP = 000BH. Now adding FFF7 (remember that the high-order byte is in the high-order address), we have

$$\begin{array}{rr} & 000BH \\ + & FFF7H \\ \hline 1 & 0002H \end{array}$$

Ignoring the final carry, the correct address (0002) is found. FFF7 is actually −9, the number of bytes to jump *backward* in 2's-complement form.

When the target address is located within −128 or +127 bytes of IP, one byte of object code can be saved by using the *short jump*. This can be seen in Fig. 2.16. The distance to jump is now (minus) eight bytes, which can be represented in one 2's complement byte as F8.

Which jump should you use—near or short? Because one less byte is required, the short form is best. In fact, when assembling your program, the assembler will automatically generate a short jump if it can determine that the target address is located within −128 to +127 bytes. The word SHORT is therefore optional. However, you are en-

```
ADDR    HEX CODES   LABELS     OP-CODE   OPERANDS      COMMENTS
0000    B3 04                  MOV       BL,04H        ;Load divisor to BL
0002    E5 06       REPEAT:    IN        AX,06H        ;Get data word
0004    F6 F3                  DIV       BL            ;Divide by four
0006    E7 9A                  OUT       9AH,AX        ;Output result
0008    E9 F7 FF               JMP       REPEAT        ;Repeat the cycle
000B
```

Figure 2.15 Assembly language program demonstrating the near jump instruction.

```
ADDR    HEX CODES  LABELS    OP-CODE     OPERANDS      COMMENTS
0000    B3 04                MOV         BL,04H        ;Load divisor to BL
0002    E5 06      REPEAT:   IN          AX,06H        ;Get data word
0004    F6 F3                DIV         BL            ;Divide by four
0006    E7 9A                OUT         9AH,AX        ;Output result
0008    EB F8                JMP SHORT   REPEAT        ;Repeat the cycle
000A
```

Figure 2.16 The program in Fig. 2.15 is rewritten using the JMP SHORT instruction. One byte of code is saved.

couraged to use it, as there may be some cases where the assembler will select a near jump when a short jump would be best. This can happen when the target's label has not yet been seen by the assembler and it must therefore assume the worst case.

The memory indirect and register indirect forms of the jump instruction specify the actual 16-bit target address. These two forms are thus not relative. As an example, we could replace the JMP SHORT REPEAT instruction in Fig. 2.16 with the two instructions

```
MOV   BX,0002H      ;BX points at REPEAT
JMP   BX            ;Jump to 0002H
```

The advantage in using the indirect jump instructions is that any address within the code segment can be specified. However, as we have just seen, an extra instruction is required to set up the target address. This is not always a disadvantage, as it allows the program the ability to compute the target address depending on program conditions.

The main advantage of using relative jumps is that the resulting program is *relocatable* anywhere within the code segment. Moving the program in Fig. 2.16 so that it begins at address E800H will not require the jump target address to be changed. It is still eight locations backward.

Finally, it is also possible to transfer control to a target address in a new code segment. This is called a *far jump*. Again direct and indirect forms are possible but neither form is relative. The direct form requires the assembler operator FAR PTR to identify the label as being in a new code segment. The indirect forms must specify a double word for the new CS and IP values.

Conditional Jump Instructions

The conditional jump instructions perform a short jump based on the condition of the status flags. The general form is shown in Table 2.14. Table 2.15 lists all the testable conditions. Usually, a conditional jump instruction is placed after an arithmetic or logical instruction, transferring control depending on the result of that instruction.

TABLE 2.15 CONDITIONAL JUMP INSTRUCTIONS

Mnemonic	Condition
	Signed Operations
JG/JNLE	Greater/not less nor equal ((SF ⊕ OF) + ZF) = 0
JGE/JNL	Greater or equal/not less (SF ⊕ OF) = 0
JL/JNGE	Less/not greater nor equal (SF ⊕ OF) = 1
JLE/JNG	Less or equal/not greater ((SF ⊕ OF) + ZF) = 1
JO	Overflow (OF = 1)
JS	Sign (SF = 1)
JNO	Not overflow (OF = 0)
JNS	Not sign (SF = 0)
	Unsigned Operations
JA/JNBE	Above/not below nor equal (CF ⊕ ZF) = 0
JAE/JNB	Above or equal/not below (CF = 0)
JB/JNAE	Below/not above nor equal (CF = 1)
JBE/JNA	Below or equal/not above (CF ⊕ ZF) = 1
	Either
JC	Carry (CF = 1)
JE/JZ	Equal/zero (ZF = 1)
JP/JPE	Parity/parity even (PF = 1)
JNC	Not carry (CF = 0)
JNE/JNZ	Not equal/not zero (ZF = 0)
JNP/JPO	Not parity/parity odd (PF = 0)

Source: J. Uffenbeck, *Microcomputers and Microprocessors: The 8080, 8085, and Z-80*. Prentice-Hall, Englewood Cliffs, N.J., 1985.

Example 2.17

Explain the operation of the following program.

```
MOV   BL,47H
IN    AL,36H
CMP   AL,BL
JE    MATCH
JA    BIG
JMP   SMALL
```

Solution The program inputs a data byte from input port 36H and then compares it with 47H. If a match occurs, control is transferred to the program beginning at address MATCH. If the input byte is >47H, control is transferred to the program beginning at address BIG. If none of these conditions are met, control is passed to the program beginning at address SMALL.

Note: MATCH, BIG, and SMALL must be located within −128 to +127 bytes of the corresponding conditional jump instruction.

The JCXZ instruction is a special conditional jump that does not test the status flags. Instead, it tests the contents of CX and transfers control to the target address if CX = 0. This has application with the LOOP instructions explained in the next section.

In summary, the conditional jump instructions are among the most important in the processor's instruction set. This is because they allow the processor to make decisions based on program conditions. You will see numerous examples throughout this book.

Loop Instructions

One common programming problem is to set up a group of instructions that must be executed several times. One of the CPU registers is loaded with the loop count and this register is decremented by 1 at the end of each loop. A JNZ (jump if not zero) instruction transfers control back to the start of the loop if the counter register is not zero.

The 8086/88 loop instructions are designed exactly for this application. They combine the decrement counter and transfer of control instructions into one. Table 2.16 lists the three forms.

Example 2.18

Using the loop instructions, write a program segment to output 256 bytes from a data table beginning at address TABLE to output port A0H.

Solution

```
        LEA     SI,TABLE        ;Load SI with base address of TABLE
        MOV     CX,0100H        ;CX holds the number of bytes to output
AGAIN:  LODSB                   ;Data byte to AL and INC SI
        OUT     0A0H,AL         ;Output the data byte
        LOOP    AGAIN           ;Repeat until CX = 0
```

Note: We have assumed that DF = 0. In Sec. 2.9 we show how to set or reset this flag.

This example shows how the string and loop instructions actually perform two operations each. LODSB (load string byte) is the equivalent of

```
MOV   AL,[SI]
INC   SI
```

and LOOP AGAIN replaces the equivalent instructions

```
DEC   CX
JNZ   AGAIN
```

The LOOPE or LOOPZ (loop if equal or zero) and LOOPNE or LOOPNZ (loop if not equal or not zero) instructions text CX and ZF. For example, LOOPE "loops while equal." This means that if CX ≠ 0 and ZF = 1, the loop will be repeated. Similarly, LOOPNE "loops while not equal." These two forms are useful when comparing two strings or looking for a match between a CPU register and a byte or word in a data table.

TABLE 2.16 LOOP INSTRUCTIONS

General mnemonic			Coding example			
Op-code	Operand	Object code	Mnemonic	Segment for memory access	Symbolic operation	Description
LOOP	*short target*	E2 —[a]	LOOP MEMS [b]	Code	CX ← CX − 1 If CX ≠ 0, then IP ← MEMS	Decrement CX and transfer control to the short target address if CX ≠ 0
LOOPE/[c] LOOPZ	*short target*	E1 —[a]	LOOPZ MEMS [b]	Code	CX ← CX − 1 If (CX ≠ 0)·(ZF = 1), then IP ← MEMS	Decrement CX and transfer control to the short target address if CX ≠ 0 AND the last instruction to affect the flags resulted in zero (ZF = 1)
LOOPNE/[c] LOOPNZ	*short target*	E0 —[a]	LOOPNZ MEMS[b]	Code	CX ← CX − 1; If (CX ≠ 0)·(ZF = 0), then IP ← MEMS	Decrement CX and transfer control to the short target address if CX ≠ 0 AND the last instruction to affect the flags resulted in a nonzero result (ZF = 0)

[a]This byte is the 2's-complement offset between short memory location MEMS and the instruction immediately following the LOOP instruction. The offset is limited to 127 locations forward (7FH) and 128 locations backward (80H).

[b]MEMS is assumed to point to a short memory location (i.e., +127 or −128 locations from the address of the instruction immediately following the LOOP instruction).

[c]Either form of the mnemonic may be used.

All forms of the loop instruction repeat until CX = 0. This means that the loop will be repeated 65,536 times if CX = 0 initially. Although the programmer normally controls the value in CX, the JCXZ instruction can be used before entering the loop "just to be sure" that CX has not somehow become zero.

Push and Pop Instructions

The PUSH and POP instructions are not really transfer-of-control instructions, but they do use the stack area of memory, which you will need to understand before studying the CALL, RET, and software interrupts in the next two sections.

Recall that the stack is a 64K-byte memory segment whose base address is determined by the SS segment register. Two CPU registers normally point into this area. These are SP and BP.

The stack is not another data segment but a special area in memory used by the processor for temporary data storage. What makes the stack unique is that it is a last-in, first-out type of memory. Just as plates are pushed into a spring-loaded dispenser in a cafeteria, data is pushed onto the stack by the PUSH *source* instruction. The POP *destination* instruction causes the data currently on top of the stack to be popped into the destination operand. Table 2.17 explains the four push and pop instructions in detail.

Figure 2.17 shows the stack segment with register SP pointing to location 1000H —currently the top of the stack. Now assume that the instruction PUSH CX occurs. SP is decremented by 2 and CL and CH are then pushed onto the stack. The new stack top is address 0FFEH (SP′). As you can see, the stack actually grows *downward* with each successive PUSH instruction. The stack top moves to 0FFCH (SP″) after the instruction PUSH BX is executed.

Now that BX and CX are safely stored in the stack segment, these registers can be used for some other purpose. When it is desired to recover the old values of BX and CX, the POP instruction is used. However, we must be careful. SP is currently pointing to location 0FFCH, where the BX register is stored. We can say that BX is "sitting atop the stack." The next POP instruction will thus cause BX to be popped into the destination operand.

Based on this observation we can state a general rule for using the stack. "Registers should be popped off the stack in the *reverse* order in which they were pushed on." In this case POP BX causes the contents of 0FFCH and 0FFDH to be popped into BL and BH and SP to be incremented by 2. CX now sits atop the stack and SP = 0FFEH (SP′). A POP CX instruction will recover CL and CH and restore SP to its original value (1000H).

Before using the stack it is important that register SP be initialized. This is accomplished with an immediate mode data move instruction such as MOV SP,1000H, which establishes the stack top at location 1000H in the stack segment. Because the stack grows downward, SP is usually initialized to a high memory location allowing room for the stack to grow.

The stack location is very important, especially if the data, code, and stack segments are all the same or allowed to overlap. If you are not careful, the stack can overwrite program data or even the program itself. This is a difficult problem to debug because there is no program left to check!

TABLE 2.17 PUSH AND POP INSTRUCTIONS

Addressing mode	Object code	Coding example: Mnemonic op-code operand	Segment for memory access	Symbolic operation	Description
PUSH *source*[a]	51	PUSH CX	Stack	SP ← SP − 2; [SP + 1] ← CH; [SP] ← CL	Decrement SP by 2 and transfer the word from the source operand to the top of the stack now pointed to by SP
	1E	PUSH DS	Stack	SP ← SP − 2; [SP + 1:SP] ← DS	
	FF 75 02	PUSH [DI + 2]	Stack, data	SP ← SP − 2; [SP + 1] ← [DI + 3]; [SP] ← [DI + 2]	
POP *dest*[a]	59	POP CX	Stack	CL ← [SP]; CH ← [SP + 1]; SP ← SP + 2	Transfer the word from the top of the stack pointed to by SP to the destination operand; increment SP by 2
	1F	POP DS	Stack	DS ← [SP + 1:SP]; SP ← SP + 2	
	8F 45 02	POP [DI + 2]	Data, stack	[DI + 3] ← [SP + 1]; [DI + 2] ← [SP]; SP ← SP + 2	
PUSHF none	9C	PUSHF	Stack	SP ← SP − 2; [SP + 1:SP] ← flags	Push the 16-bit flag word onto the stack
POPF none	9D	POPF	Stack	flags ← [SP + 1:SP]; SP ← SP + 2	Pop the top of the stack into the 16-bit flag word

[a]Immediate operands are not allowed (except with the 80186).

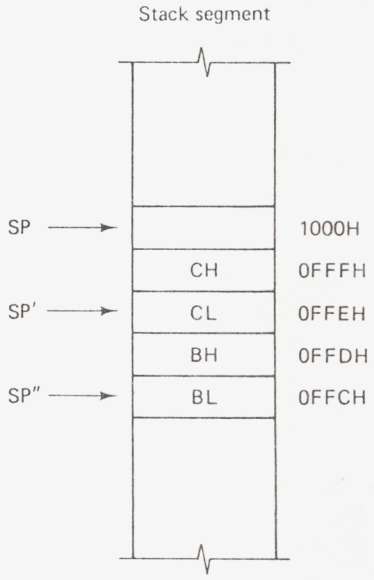


Figure 2.17 Contents of the stack and register SP after the PUSH CX and PUSH BX instructions have executed. SP'' is the final value of register SP.

Finally, note that the PUSHF and POPF instructions allow the processor flags to be stored on the stack. This can be useful with interrupts and subroutines as the entire state of the machine (all CPU registers and flags) can be saved and then restored later. These concepts are explored in later chapters.

Call and Return Instructions

The CALL and RET instructions allow the programmer to use a group of instructions as a *subroutine* or *procedure* that can be executed several times from within the main program. As an example, the following instructions will generate a short time delay.

```
DELAY   PROC    NEAR    ;Define near procedure
        PUSH    AX      ;Save AX on stack
        MOV     AX,0000 ;Set counter for 65536 cycles
REPEAT: DEC     AX      ;Bump counter
        JNZ     REPEAT  ;Do 65536 times
        POP     AX      ;Restore AX
        RET             ;Return to main program
DELAY   ENDP
```

The first line defines to the assembler the following group of instructions as a *near* procedure (i.e., a subroutine within the segment) and gives it the name DELAY. The subroutine itself kills time by repeating the DEC AX and JNZ REPEAT instructions 65536 times. This will require approximately 250 ms with a 5-MHz system clock. Once the time delay has ended, the value of AX before the procedure began is popped back into AX and control is transferred back to the main program via the RET instruction. The last line tidies things up, telling the assembler that this is the end of the procedure.

The DELAY subroutine can be called by any program in which a 250-ms time delay is required. This might be required when outputting data to a peripheral that cannot

accept data as fast as the microprocessor can output it, for example. Another application would be waiting for a mechanical switch to stop bouncing before testing its value. The calling procedure would look as follows:

```
main program
.
.
.
CALL DELAY
(main program continues)
.
.
.
```

The CALL instruction is similar to an unconditional jump except that the value of IP (now pointing to the instruction immediately following the CALL) is pushed onto the stack. Control then transfers to address DELAY.

After the DELAY subroutine has executed, it must end with a RET (return) instruction. This causes the top of the stack to be popped into IP, neatly returning control to the instruction in the main program with which it left off.

Table 2.18 describes the various forms of the CALL and RET instructions. The CALL we have just been discussing is referred to as a *near* CALL because the subroutine is located within the same code segment. It is also possible to execute a *far* CALL to a procedure in a different code segment.

The far CALL differs from the near CALL in that the value of CS (in addition to IP) is saved on the stack. For this reason you must also use a far RET instruction to pop CS and IP from the stack when the far procedure has ended. The assembler will automatically generate the proper RET instruction if you have defined the procedure using the NEAR or FAR attribute.

Like the near jump instructions, the near CALL can be direct or indirect. The direct forms are relative, allowing the subroutine to be located within +32767 to −32768 bytes of the address in IP. There are no short forms. The indirect forms specify the absolute address in a memory location or CPU register.

A direct far CALL requires the operator FAR PTR to tell the assembler that the subroutine is located in another segment. The indirect forms require a double word to specify the new CS and IP values.

You will note in Table 2.18 that the RET instructions can specify an optional *pop value*. This allows data to be passed to a subroutine on the stack instead of in a CPU register: for example,

```
        PUSH  CX
        PUSH  BX
        CALL  SUB
SUB:    .
        .
        .
        RET   4
```

TABLE 2.18 CALL AND RETURN INSTRUCTIONS

General mnemonic		Object code	Coding example			
Op-code	Operand		Mnemonic	Segment for memory access	Symbolic operation	Description
Call	*near target*	E8 — —[a]	CALL MEMN[b]	Code	SP ← Sp − 2; [SP + 1:SP] ← IP; IP ← MEMN	IP is pushed onto the top of the stack and control is transferred within the segment to the near target address
		FF 16 00 10	CALL [MEMWDS][c]	Data	SP ← SP − 2; [SP + 1:SP] ← IP IP ← [1001H:1000H]	
		FF 15	CALL [DI]	Data	SP ← SP − 2; [SP + 1:SP] ← IP; IP ← [DI + 1:DI]	
		FF D7	CALL DI	Within CPU	SP ← SP − 2 [SP + 1:SP] ← IP; IP ← DI	
CALL	*far target*	9A 00 10 D3 09	CALL FAR PTR MEMF[d]	Code	SP ← SP − 2; [SP + 1:SP] ← CS; CS ← 09D3H; SP ← SP − 2; [SP + 1:SP] ← IP; IP ← 1000H	CS and IP are pushed onto the top of the stack and control is transferred to the new segment and far target address
		FF 1E 00 10	CALL [MEMWWDS][e]	Data	Same as above except: CS ← [1003H:1002H]; IP ← [1001H:1000H]	
		FF 1D	CALL DWORD PTR[DI]	Data	Same as above except: CS ← [DI + 3:DI + 2]; IP ← [DI + 1:DI]	

TABLE 2.18 (continued)

General mnemonic		Object code	Coding example			
Op-code	Operand		Mnemonic	Segment for memory access	Symbolic operation	Description
RET	*n* (near)[f]	C3	RET	Stack	IP ← [SP + 1:SP]; SP ← SP + 2	The word at the top of the stack is popped into IP transferring control to this new address; RET is normally used to return control to the instruction following a near subroutine call; if included, the optional pop value (n) is added to SP
		C2 08 00	RET 8	Stack	IP ← [SP + 1:SP]; SP ← SP + 2 + 8	
RET	*n* (far)[f]	CB	RET	Stack	IP ← [SP + 1:SP]; SP ← SP + 2; CS ← [SP + 1:SP]; SP ← SP + 2	As above except that the double word at the top of the stack is popped into IP and CS transferring control to this new far address
		CA 08 00	RET 8	Stack	IP ← [SP + 1:SP]; SP ← SP + 2; CS ← [SP + 1:SP]; SP ← SP + 2 + 8	

[a]These two bytes are the 2's-complement offset between near memory location MEMN and the instruction immediately following the CALL instruction. For example, if the CALL is to be 12 locations forward, the offset is 00 0C. Similarly, 12 locations backward requires an offset of FFF4.

[b]MEMN is assumed to point to a near memory location (i.e., within the segment).

[c]MEMWDS is assumed to point to the word beginning at location 1000H in the data segment. The brackets ae optional because MEMWDS defines a word not a location to jump to.

[d]FAR PTR indicates that MEMF is located in a different code segment. In this case it is assumed that MEMF points at location 09D3H:1000H.

[e]MEMWWDS is assumed to point to the double word beginning at location 1000H in the data segment. The brackets are optional because MEMWWDS defines a double word, not a location to jump to.

[f]The same mnemonic is used for a near or far return. The assembler will generate the proper code based on the procedure definition statement (near or far).

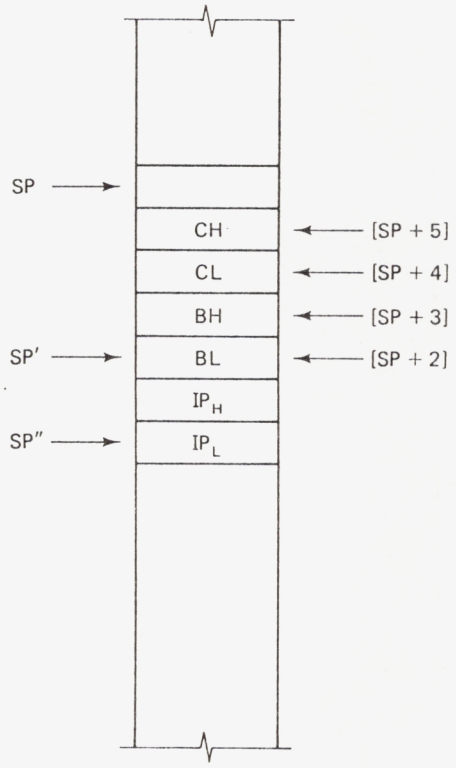


Figure 2.18 Passing parameters to a near subroutine on the stack. In this example four bytes are passed by the calling routine.

The main program pushes four bytes (registers CX and BX) onto the stack before calling the subroutine. This data can then be accessed by the subroutine using register BP to point to locations SP+2 through SP+5 (assuming a near call) as shown in Fig. 2.18. When the subroutine has ended and the data is no longer important, the RET 4 instruction pops IP off the stack, incrementing SP to SP′. The pop value of 4 then causes SP′ to be incremented to SP, its original value before the data was pushed onto the stack. This effectively deletes the data, as it will be overwritten with the next stack operation.

Software Interrupts

An interrupt is a request of the processor to suspend its current program and transfer control to a new program called the *interrupt service routine* (ISR). The interrupt request can be initiated in hardware or software. For the 8086/88, applying a logic 1 to the INTR or NMI input lines will initiate the interrupt request. In Chap. 8 we discuss in detail the processing of hardware interrupts.

Software interrupts are initiated by giving the instruction INT *type*. The processor responds by pushing the flags, CS, and IP onto the stack. This is shown in Fig. 2.19. It then loads new values for CS and IP from an interrupt jump table located in absolute memory from 00000 to 003FFH. These 1K bytes allow 256 different software interrupt requests. This is shown in Fig. 2.20.

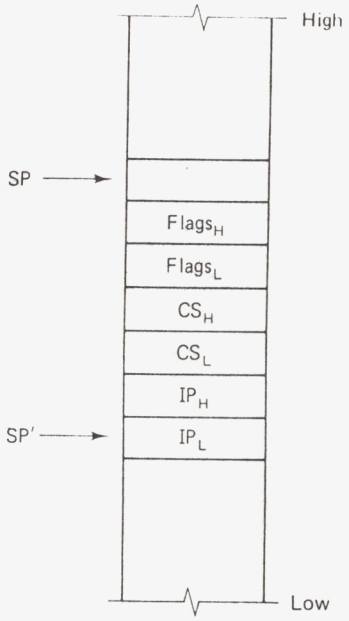


Figure 2.19 Stack area after executing an INT *type* instruction. SP′ is the new value of register SP.

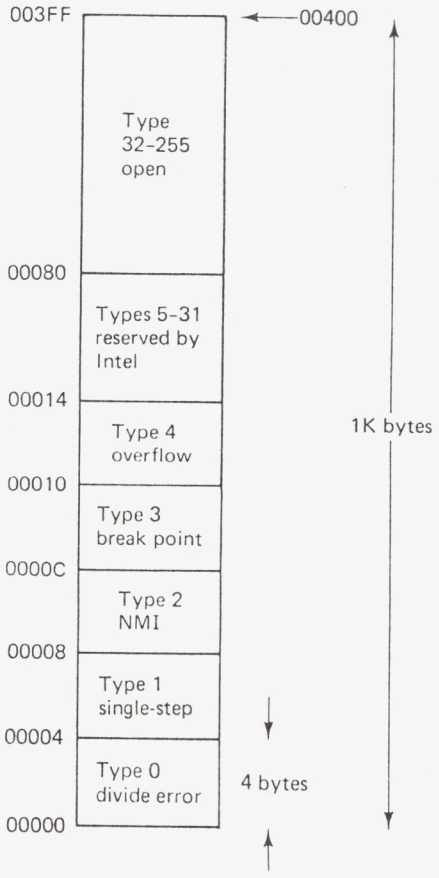


Figure 2.20 The 8086/88 software interrupt jump table will accommodate 256 interrupt types. (From J. Uffenbeck, *Microcomputers and Microprocessors: The 8080, 8085, and Z-80*. Prentice-Hall, Englewood Cliffs, N.J., 1985.)

Table 2.19 provides the specific details for the software interrupt instructions. If an INT 23H is executed, the processor will multiply 23H by 4 (rotate left twice), resulting in the jump table address 0008CH.

$$00100011 \times 4 = 10001100 = 0008CH$$

CS and IP will then be loaded with the double word stored in 0008CH through 0008FH (CS in the high-order word, IP in the low-order word). In effect, INT 23H performs a far call to the address stored in the double word at 0008CH through 0008FH.

When the interrupt service routine has finished, the IRET (interrupt return) instruction should be executed. This instruction is similar to the far return instruction but also pops the flag register from the stack (in addition to CS and IP).The main program then resumes with the instruction immediately following INT 23H.

Because a hardware interrupt can also generate an INT 23H, one of the uses for software interrupts is to debug interrupt service routines without having to build or use the hardware to request the interrupt.

Some of the interrupt jump table locations are reserved for special functions, as shown in Fig. 2.20. For example, whenever a divide-by-zero error occurs, the double word in 00000–00003H is used to determine the address of the error response routine. Similarly, if TF is set control transfers to the routine whose address is stored in locations 00004H–00007H after the execution of each main program instruction.

The INTO (interrupt on overflow) instruction is special in that no type number need be specified. After performing a signed arithmetic instruction, executing INTO will generate a type 4 interrupt if an overflow condition has occurred.

Note that executing an interrupt instruction (either software or hardware initiated) will clear IF and TF. This ensures that the interrupt service routine is not single-stepped and also blocks external (maskable) interrupts on INTR. Nonmaskable interrupt requests on NMI will still be accepted, as will other software interrupts. Executing the IRET instruction will restore the flags to their original values before the interrupt occurred. Thus you need not be concerned about setting IF or TF before returning from the interrupt service routine. The use of interrupts is explored more fully in later chapters.

SELF-REVIEW 2.8 (answers on page 108)

2.8.1. List the five instruction types that can be used to transfer control to a new (nonsequential) address.

2.8.2. What is the difference between a short, near, and far memory location?

2.8.3. All direct jumps or procedure calls are relative, but the indirect forms must specify the absolute address. (True/False)

2.8.4. What is the advantage in using *relative* jump or procedure calls versus the absolute address forms?

2.8.5. The instruction JMP DWORD PTR CS:[BP] is an example of a(n) _____ jump.

2.8.6. In question 2.8.5 the target address is stored in the _____ segment.

2.8.7. The instruction codes below will transfer control to location ZERO if the data input is _____.

Table 2.19 SOFTWARE INTERRUPT INSTRUCTIONS

General mnemonic		Object code	Coding example			
Op-code	Operand		Mnemonic	Segment for memory access	Symbolic operation	Description
INT	*type*	CD 23	INT 23H	Stack and interrupt jump table at 00000–003FFH	SP ← SP − 2; [SP + 1:SP] ← flags IF ← 0; TF ← 0; SP ← SP − 2; [SP + 1:SP] ← CS; CS ← [0008FH:0008EH];[a] SP ← SP − 2 [SP + 1:SP] ← IP; IP ← [0008DH:0008CH][b]	Save the flag, CS, and IP registers on the stack and transfer control to the far address stored in the double word beginning at absolute address *type* * 4
INTO	none	CE	INTO	Stack and interrupt jump table at 00000–003FFH	If OF = 1, then SP ← SP − 2 [SP + 1:SP] ← flags; IF ← 0; TF ← 0; SP ← SP − 2; [SP + 1:SP] ← CS; CS ← [00013H:00012H];[a] SP ← SP − 2; [SP + 1:SP] ← IP; IP ← [00011H;00010H][b]	If an overflow condition exists (OF = 1), a type 4 interrupt is executed
IRET	none	CF	IRET	Stack	IP ← [SP + 1:SP]; SP ← SP + 2; CS ← [SP + 1:SP]; SP ← SP + 2; flags ← [SP + 1:SP]; SP ← SP + 2	Transfer control back to the point of interrupt by popping the IP, CS, and flag registers from the stack; IRET is normally used to exit any interrupt procedure whether activated by hardware or software

[a][(type * 4) + 3: (type * 4) + 2].

[b][(type * 4) + 1: (type * 4)].

```
INPUT:  IN    AL,80H
        DEC   AL
        JNZ   INPUT
        JMP   ZERO
```

2.8.8. The LOOP instruction is the equivalent of the two instruction types _____ and _____.

2.8.9. As data is pushed onto the stack, the stack grows _____ in memory.

2.8.10. How many bytes are pushed onto the stack by a far CALL instruction? What do they represent?

2.8.11. After executing the instruction INT 80H, from which memory location will the next instruction be fetched?

PROCESSOR CONTROL INSTRUCTIONS

The final group of instructions to be discussed are those used to control the operation of the processor and set or clear the status indicators. These are shown in Table 2.20.

The carry, direction, and interrupt flags can each be set or cleared, and the carry flag can also be *inverted*. The remaining status flags (PF, AF, ZF, and OF) cannot be accessed with specific control instructions. However, their values are determined by the results of particular arithmetic and logical instructions.

DF, IF, and TF are processor control bits rather than status indicators. DF is used with the string group of instructions to determine if the pointer registers are to be incremented or decremented. The STD (set direction flag) and CLD (clear direction flag) instructions are used to set or clear this flag.

STI (set interrupt enable flag) and CLI (clear interrupt enable flag) enable or disable maskable interrupts on the INTR input line. Clearing this bit blocks all interrupts on INTR effectively *masking* this input.

The third processor control bit is TF, the trap flag. When set, a type 1 interrupt is generated after completion of each processor instruction. There is no instruction for setting or resetting this flag (Intel design error?), but the following sequence of instructions can be used (in this example) to set TF. Refer to Fig. 2.21 for the flag bit positions.

```
PUSHF                          ;Copy flags onto stack
MOV   BP,SP                    ;Point BP at stack top
OR    BYTE PTR[BP+1],01H       ;Set bit 0—TF
POPF                           ;Restore flags
```

The HALT instruction will stop the processor and cause it to enter an idle loop. However, once halted it can be restarted only via a hardware interrupt or a system reset. For this reason, ending your programs with a HALT instruction may not be a good idea (restarting the computer may erase your program).

The NOP instruction may seem rather useless; after all, it is a "no operation" instruction. However, during debugging it may be useful to "NOP out" several instructions to avoid a long loop or other program condition. Later, the original instructions can be replaced. NOPs may also be used to "pad" time-delay routines.

TABLE 2.20 PROCESSOR CONTROL INSTRUCTIONS

General mnemonic		Object code	Coding example			
Op-code	Operand		Mnemonic	Segment for memory access	Symbolic operation	Description
STC	None	F9	STC	Within CPU	CF ← 1	Set carry flag
CLC	None	F8	CLC	Within CPU	CF ← 0	Clear carry flag
CMC	None	F5	CMC	Within CPU	CF ← $\overline{CF}$	Complement carry flag
STD	None	FD	STD	Within CPU	DF ← 1	Set direction flag (auto decrement for string instructions)
CLD	None	FC	CLD	Within CPU	DF ← 0	Clear direction flag (auto increment for string instructions)
STI	None	FB	STI	Within CPU	IF ← 1	Set interrupt flag (enabling interrupts on the INTR line)
CLI	None	FA	CLI	Within CPU	IF ← 0	Clear interrupt flag (disabling interrupts on the INTR line)
HLT	None	F4	HLT	Within CPU	None	Halt
WAIT	None	9B	WAIT	Within CPU	None	Enter wait state if $\overline{TEST}$ line = 1
LOCK	*instruction*	F0 A1 00 10	LOCK MOV AX,MEMWDS[a]	Data	None	Output line $\overline{LOCK}$ = 0 while the instruction following LOCK executes; used to prevent coprocessors from accessing the bus during a particular instruction
NOP	None	90	NOP	Data	None	No operation
ESC	*number, source*	DE 0E 00 10	ESC 31H,MEMWDS[a]	Data	Data bus ← [MEMWDS]	Place the contents of the memory source operand on the data bus and execute a NOP; the first operand identifies a particular escape instruction to be executed by a coprocessor

[a]MEMWDS is assumed to point at the word beginning at location 1000H in the data segment.

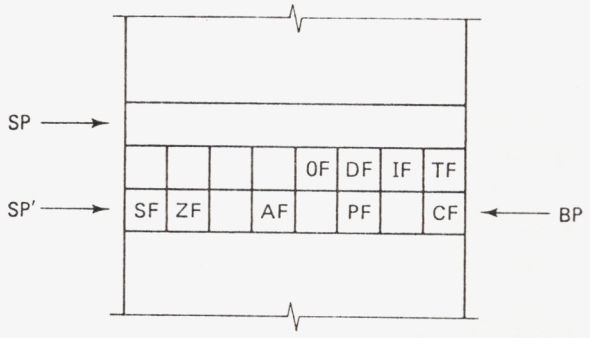


Figure 2.21 Contents of the stack after executing PUSHF. TF is bit 0 of [SP + 1].

Three processor control instructions are intended for use with special *coprocessors*. The WAIT instruction is used to synchronize the 8086/88 with the 8087 NDP via the $\overline{\text{TEST}}$ input. When the 8087 is busy executing an instruction and cannot accept new data or instructions from the 8086/88, it will drive this input high. By preceding each instruction that requires data from the 8087 with a WAIT instruction, the 8086/88 ensures that data will not be lost between these two processors.

The LOCK instruction is executed by the 8086/88 when it is important that no other processor attempt to control the system buses. LOCK is in effect for the single instruction only. It prevents a coprocessor from changing or accessing a memory location that the 8086/88 is about to access.

The ESC (escape) instruction is used as a prefix to the coprocessor instructions. The 8086/88 will place the source operand on the data bus but will take no further action. The coprocessor, constantly examining the data bus contents, is activated by the ESC prefix, causing it to read the two operands and begin execution. The 8087 and 8089 coprocessors are discussed in Chap. 11.

REVIEW 2.9 (answers on page 108)

2.9.1. To be sure that the carry flag is 0 before the execution of a particular set of instructions, the _____ instruction should be used.

2.9.2. Executing the instruction STD will affect only the string instructions. (True/False)

2.9.3. The 8086/88 has three instructions for controlling a coprocessor. These are _____

_____, and _____.

CHAPTER 2 SELF-TEST

1. Under what conditions will the contents of the queue hold the "wrong" op-codes?
2. Because of its pipelined architecture, the 8086 can execute several instructions at the same time. (True/False)
3. List all 8086/88 registers that can be accessed as bytes or as words.
4. Because of its segmented memory, the 8086/88 requires a minimum of 4 × 64K or 256K bytes of memory. (True/False)
5. Which of the following addresses are valid base addresses for one of the segment registers? (a) FFFFFH; (b) A0FF0H; (c) 90000H; (d) 10008H.

6. How would you tell the assembler to initialize a memory word TEMP with 0000H?
7. What is the logical code segment address corresponding to physical address 6A3F7H? Assume that CS = 5C00H.
8. Which addressing mode would you use to initialize a CPU register with data?
9. What is the advantage of the register indirect addressing modes versus the direct addressing mode?
10. Which CPU registers, when used as memory pointers, cannot have their default memory segment assignments overridden?
11. Each of the following instructions is invalid: (1) MOV DS,1234H; (2) IN AL,70FFH; (3) MOV CX,[AX]. Explain why.
12. Which string instruction should be used to ensure that two strings in memory are equal?
13. Which instruction should be given to ensure that the memory pointers in question 12 will automatically increment?
14. What is the difference between a rotate and a shift instruction?
15. To mask bits 4–7 of register CH and force these bits low, the instruction _____ should be used.
16. When performing _____ _____ arithmetic, the ADC and SBB instructions should be used.
17. If AL = 09H and BL = 08H, the instruction sequence

```
MUL  BL
AAM
```

will store _____ in register AX.

18. All conditional jump instructions perform _____ jumps.
19. To repeat a string instruction while a particular condition is zero, use the _____ instruction.
20. To transfer control to the address stored in the double word beginning at address 00084H, the software interrupt instruction _____ should be used.

ANALYSIS AND DESIGN QUESTIONS

2.1. The Intel manual for the 8086 indicates that "a series of fast-executing instructions can drain the queue and increase instruction execution time." Explain this statement.

2.2. Determine the contents of register AL,BL and the six status flags after the following instructions are executed.

```
STC
MOV  AL,4CH
SBB  AL,3EH
XOR  BL,BL
MOV  [SI],BL
```

2.3. Determine the values of the 8086's six status flags after the program in Ex. 2.12 has executed.

2.4. Determine the value of register SP after the following instructions are executed.

```
MOV     SP,0FFFFH
PUSHF
PUSH    CX
CALL    DELAY
POP     CX
```

2.5. Using the segment register assignments shown in Fig. 2.9, what is the physical address of the memory operand for the following AND instruction?

```
MOV   BP,1080H
MOV   BX,007FH
AND   BYTE PTR[BP + BX + 30H],80H.
```

2.6. Write the programming sequence required to load the extra and data segment registers with 80A0H.

2.7. What is "wrong" with the instruction REP LODSB?

2.8. Write a program to input a data byte from I/O port 8800H and test bit 1. If this bit is low, transfer control to location START; if high, repeat the sequence.

2.9. Describe the output produced by the following program:

```
        MOV     AL,0
REPEAT: OUT     0,AL
        INC     AL
        DAA
        JMP     REPEAT
```

2.10. What single 8086/88 instruction is the equivalent of the following four instructions?

```
PUSH  BX
PUSH  AX
POP   BX
POP   AX
```

2.11. The following program is written to compare two strings stored in memory. However, it contains at least two "bugs." What do you think is wrong with the program?

```
        LEA     SI, STRING1
        LEA     DI,STRING2
        CLD
        MOV     CX,NO_OF_BYTES
REPNE   SCASB
        JNZ     ERROR
        JMP     OK
```

8 Debug

In this chapter, you will learn

- How to start the **debug** utility
- How to use the **debug** commands and parameters

Note You need to read this chapter only if you are writing or testing executable programs (*.com* or *.exe* files) or object (*.obj*) files.

Introduction

The **debug** utility is a debugging program that provides a controlled testing environment for binary and executable object files. Note that **Edlin**, the MS-DOS line editor, is used to alter source files; **debug** is **Edlin**'s counterpart for binary files.

Debug eliminates the need to reassemble a program to see if a problem has been fixed by a minor change. It allows you to alter the contents of a file or the contents of a CPU register, and then immediately reexecute a program to check the validity of the changes made.

All **debug** commands may be aborted at any time by pressing CONTROL-C. The CONTROL-S key sequence suspends the display, so that you can read it before the output scrolls away. Pressing any key other than CONTROL-C or CONTROL-S restarts the display. All these commands are consistent with the control character functions available at the MS-DOS command level.

How to Start Debug

Starting Debug

Debug may be started two ways. By the first method, you type all commands in response to the **debug** prompt (a hyphen). By the second method, you type all commands on the line used to start **debug**.

Method 1: **debug**
Method 2: **debug** [*filename* [*arglist*]]

Method 1

Method 1: Debug

To start **debug** using method 1, simply type the following:

```
debug
```

Debug responds with the hyphen (-) prompt, signaling that it is ready to accept your commands. Since you didn't specify a filename, you can use other commands to work on current memory, disk sectors, or disk files.

Warnings

- When **debug** (version 3.0) is started, it sets up a program header at offset 0 in the program work area. In previous versions of **debug**, you could overwrite this header. You can still overwrite the default header if you don't give a *filename* to **debug**. If you are debugging a *.com* or *.exe* file, however, do not tamper with the program header below address 5CH, or **debug** will terminate.
- Do not restart a program after the following message is displayed:

  ```
  Program terminated normally
  ```

  You must reload the program with the N (name) and L (load) commands for it to run properly.

Copied pursuant to S53B of the Copyright Act as amended on behalf of Murdoch Uni. on 17.2.89

DEBUG COMMANDS H1
Extract from: Microsoft MS-DOS User's Guide and User's Reference p223-256, Microsoft Corporation 1987 pp 386. ISBN - not given in original.

Method 2: Command Line

To start **debug** using a command line, you must use the following syntax:

debug [*filename* [*arglist*]]

For example, if you specify a *filename*, the following would be a typical command to start **debug**:

```
debug file.exe
```

Debug would then load *file.exe* into memory starting at 100 (hexadecimal) in the lowest available segment. The **BX:CX** registers are loaded with the number of bytes placed into memory.

If you do include a *filename*, you might also specify an *arglist*. An *arglist* is a list of filename parameters and switches that are to be passed to the program *filename*. So when *filename* is loaded into memory, it is loaded as if it had been started with a command of the form, **debug** *filename arglist*.

Here, *filename* is the file to be debugged, and *arglist* is the rest of the command line used when **debug** calls and loads *filename* into memory.

Debug Command Information

Each **debug** command consists of a single letter followed by one or more parameters. Additionally, the control characters and special editing functions described in Chapter 5, "MS-DOS Editing and Function Keys," apply to **debug** as well.

If a syntax error occurs in a **debug** command, **debug** reprints the command line and indicates the error with a caret (^) and the word "Error" as in the following example,

```
dcs:100 cs:110
   ^ Error
```

Note that when typing commands and parameters you may use any combination of uppercase and lowercase letters.

The **debug** commands are listed. Following this list, the commands and their parameters are described in greater detail.

Method 2

About Debug commands

How to use the Debug command parameters

Debug command	Function
A [*address*]	Assemble
C *range address*	Compare
D [*range*]	Dump
E *address* [*list*]	Enter
F *range list*	Fill
G [=*address* [*address...*]]	Go
H *value value*	Hex
I *value*	Input
L [*address* [*drive:record record*]]	Load
M *range address*	Move
N *filename* [*filename*]	Name
O *value byte*	Output
Q	Quit
R [*register-name*]	Register
S *range list*	Search
T [=*address*] [*value*]	Trace
U [*range*]	Unassemble
W [*address* [*drive:record record*]]	Write

Debug Command Parameters

All **debug** commands accept parameters, except the Q (quit) command. Parameters may be separated by delimiters (spaces or commas), but a delimiter is required only between two consecutive hexadecimal values. Thus, the following commands are equivalent:

```
dcs:100 110
d cs:100 110
d,cs:100,110
```

Parameter	Definition
drive:	A one-digit hexadecimal value that indicates which drive a file will be loaded from or written to. The valid values are 0–3, where 0 = A:, 1 = B:, 2 = C:, 3 = D:.
byte	A two-digit hexadecimal value placed in or read from an address or register.
record	One-digit to three-digit hexadecimal value that indicates the logical record number on the disk and the number of disk sectors to be written or loaded. Logical records correspond to sectors; however, since they represent the entire disk space, their numbering differs.
value	A hexadecimal value of up to four digits that specifies a port number or the number of times a command should repeat its functions.

address	A two-part designation containing either an alphabetic segment register or a four-digit segment address plus an offset value. You may omit the segment name or segment address, in which case the default segment DS is used for all commands except G, L, T, U, and W, for which the default segment is CS. All numeric values are hexadecimal.
	Following is an example *address*:
	`CS:0100`
	`04BA:0100`
	Note that the colon is required between the segment name (whether numeric or alphabetic) and the offset value.

range	Contains two addresses: for example, *address address*; or one address, an L, and a value: for example, *address* L *value* where *value* is the number of lines on which the command should operate (L80 is assumed). The second type of *range* cannot be used if another hexadecimal value follows, since the hexadecimal value would be interpreted as the second *address* of the *range*.
	Here are some example ranges:
	`CS:100 110`
	`CS:100 L 10`
	`CS:100`
	The following example, however, is illegal:
	`CS:100 CS:110`
	`b Error`
	The limit for *range* is 10000 (hexadecimal). To specify a *value* of 10000 with only four digits, type *0000* (or *0*).

list	A series of *byte* values or *strings*. *List* must be the last parameter on the command line. Following is an example *list*: `fcs:100 42 45 52 54 41`

string	Any number of characters enclosed in quotation marks. The quotation marks may be either single (' ') or double (" "). If the delimiter marks must appear within a *string*, you must use the double quotation marks.
	For example, the following strings are legal:
	`"This 'string' is okay."`
	`"This ""string"" is okay."`
	However, this string is illegal:
	`""This "string" is not okay.""`
	Note that the double quotation marks are not necessary in the following strings:
	`"This ''string'' is not necessary."`
	`'This ""string"" is not necessary.'`
	The ASCII values of the characters in the string are used as a *list* of byte values.

Assemble

(A)ssemble

Purpose:

Assembles 8086/8087/8088 mnemonics directly into memory.

Syntax:

A[*address*]

Comments:

If it finds a syntax error, **debug** responds with the following message, then redisplays the current assembly address:

```
^Error
```

All numeric values are hexadecimal and you must type them as 1–4 characters. Also, you must specify prefix mnemonics *in front of* the opcode to which they refer. You may type them on a separate line, however.

The segment override mnemonics are **CS:**, **DS:**, **ES:**, and **SS:**. The mnemonic for the far return is **RETF**. String manipulation mnemonics must explicitly state the string size. For example, use **MOVSW** to move word strings, and use **MOVSB** to move byte strings.

The assembler will automatically assemble short, near, or far jumps and calls, depending on byte displacement, to the destination address. You may override these jumps and calls by using a **NEAR** or **FAR** prefix, as in the following example:

```
0100:0500 JMP   502        ; a 2-byte short jump
0100:0502 JMP   NEAR 505   ; a 3-byte near jump
0100:505  JMP   FAR  50A   ; a 5-byte far jump
```

You may abbreviate the **NEAR** prefix to **NE**, but the **FAR** prefix cannot be abbreviated.

Debug cannot tell whether some operands refer to a word memory location or to a byte memory location. In this case, the data type must be explicitly stated with the prefix, **WORD PTR** or **BYTE PTR**. Acceptable abbreviations are **WO** and **BY**. For example,

```
NEG     BYTE PTR [128]
DEC     WO [SI]
```

Debug also cannot tell whether an operand refers to a memory location or to an immediate operand, so it uses the common convention that operands enclosed in square brackets refer to memory locations. For example,

```
MOV     AX,21    ; Load AX with 21H
MOV     AX,[21]  ; Load AX with the
                 ; contents
                 ; of memory location 21H
```

Two popular pseudo-instructions are available with the A (assemble) command: the **DB** opcode, which assembles byte values directly into memory; and the **DW** opcode, which assembles word values directly into memory. Following are examples of both:

```
DB      1,2,3,4,"THIS IS AN EXAMPLE"
DB      'THIS IS A QUOTATION MARK: "'
DB      "THIS IS A QUOTATION MARK: '"

DW      1000,2000,3000,"BACH"
```

The **A** command supports all forms of register indirect commands. For example,

```
ADD     BX,34[BP+2].[SI-1]
POP     [BP+DI]
PUSH    [SI]
```

All opcode synonyms are also supported, as in the next example:

```
LOOPZ   100
LOOPE   100

JA      200
JNBE    200
```

For 8087 opcodes, the **WAIT** or **FWAIT** prefixes must be explicitly specified, as in these last examples:

```
FWAIT FADD ST,ST(3)      ; This line assembles
                         ; an FWAIT prefix
LD TBYTE PTR [BX]        ; This line does not
```

Compare

Purpose:

Compares the portion of memory specified by *range* to a portion of the same size beginning at the specified *address*.

Syntax:

Crange address

Comments:

If the two areas of memory are identical, there is no display, and **debug** returns with the MS-DOS prompt. If there *are* differences, they are displayed in this format:

address1 byte1 byte2 address2

Example:

The following commands have the same effect:

```
C100,1FF 300
```

or:

```
C100L100 300
```

Each command compares the block of memory from 100 to 1FFH with the block of memory from 300 to 3FFH.

(C)ompare

Comparing blocks of memory

(D)ump

Dump

Purpose:

Displays the contents of the specified *range* of memory.

Syntax:

D[*range*]

Comments:

If you specify a *range* of addresses with the **D** (dump) command, the contents of the *range* are displayed. If you don't use parameters with the **D** command, 128 bytes are displayed at the first address (DS:100) after the address displayed by the previous **D** command.

The dump is displayed in two portions: a hexadecimal dump (each byte is shown in hexadecimal value) and an ASCII dump (the bytes are shown in ASCII characters). Nonprinting characters are denoted by a period (.) in the ASCII portion of the display. Each display line shows 16 bytes, with a hyphen between the eighth and ninth bytes. Each displayed line begins on a 16-byte boundary. At times in this chapter, displays are split to fit them on the page.

Examples:

If you type the command

```
dcs:100 110
```

debug displays the dump in the following format:

```
04BA:0100 42 45 52 54 41 ... 4E 44 TOM SAWYER
```

If you simply type the **D** command, the display is formatted as just described. Each line of the display begins with an address incremented by 16 from the address on the previous line.

Each subsequent **D** (typed without parameters) displays the bytes immediately following those last displayed.

Using the Dump command

If you type the following command, the display is formatted as described above, but 20H bytes are displayed:

```
DCS:100 L 20
```

If you then type the following command, the display is formatted as described above, but all the bytes in the range of lines from 100H to 115H in the CS segment are displayed:

```
DCS:100 115
```


(E)nter

Enter

Purpose:

Enters byte values into memory at the specified *address*.

Syntax:

E*address*[*list*]

Comments:

If, when using the **E** (enter) command, you type the optional *list* of values, the byte values are replaced automatically. (If an error occurs, no byte values are changed.)

If you type the *address* without the optional *list*, **debug** displays the *address* and its contents, repeats the *address* on the next line, and then waits for your input. At this point, the E command waits for you to perform one of the following actions:

- Replace a byte value with a value you type. Simply type the value after the current value. If the one you type is not a legal hexadecimal value or if it contains more than two digits, the illegal or extra character is not echoed.
- Press the SPACEBAR to advance to the next byte. To change the value, simply type the new value as described in the previous action. If, when you press the SPACEBAR, you move beyond an 8-byte boundary, **debug** starts a new display line with the address displayed at the beginning.
- Type a hyphen (-) to return to the preceding byte. If you decide to change a byte behind the current position, typing the hyphen returns the current position to the previous byte. When you type the hyphen, a new line is started with its address and byte value displayed.
- Press the RETURN key to terminate the **E** command. The RETURN key may be pressed at any byte position.

Examples:

Suppose you type the following command:

```
ECS:100
```

Now suppose that **debug** displays the following:

```
04BA:0100 EB._
```

To change this value to, say, 41, type the number 41 at the cursor, as shown:

```
04BA:0100 EB.41_
```

To step through the subsequent bytes, you would press the SPACEBAR until you saw the following:

```
04BA:0100 EB.41  10. 00.  BC._
```

To change BC to the number 42, for instance, you would type the number at the cursor, as follows:

```
04BA:0100 EB.41  10. 00.  BC.42_
```

Notice that the value 10 should be 6F. To correct this value, you would type the hyphen as many times as needed to return to byte 0101 (value 10), then replace 10 with 6F:

```
04BA:0100  EB.41    10.  00.   BC.42-
04BA:0102  00.-_
04BA:0101  10.6F_
```

Pressing the RETURN key ends the **E** command and returns you to the **debug** command level.

Using the Enter command

(F)ill

Fill

Purpose:

Fills the addresses in the specified *range* with the values in the specified *list*.

Syntax:

F*range list*

Comments:

If the *range* contains more bytes than the number of values in the *list*, the *list* will be used repeatedly until all bytes in the *range* are filled.

If the *list* contains more values than the number of bytes in the *range*, the extra values in the *list* are ignored. If any of the memory in the *range* is not valid (bad or nonexistent), the error will occur in all succeeding locations.

Example:

Suppose you type the following command:

```
F04BA:100 L 100 42 45 52 54 41
```

In response, **debug** would fill memory locations 04BA:100 through 04BA:1FF with the bytes specified. The five values would then be repeated until all the 100H bytes were filled.

Using the Fill command

Go

Purpose:

Executes the program currently in memory.

Syntax:

G[=*address* [*addresses*]]

Comments:

If you type the G (go) command by itself, the program currently in memory executes as if it had run outside **debug**.

If you set =*address*, execution of the **G** command begins at the address specified. The equal sign (=) is required so that **debug** can distinguish the start =*address* from the breakpoint *addresses*.

With the other optional *addresses* set, execution stops at the first *address* encountered, regardless of that address' position in the list of addresses that halt execution or program branching. When program execution reaches a breakpoint, the registers, flags, and decoded instruction are displayed for the last instruction executed. The result is the same as if you had typed the **R** (register) command for the breakpoint address.

You may set up to ten breakpoints, but only at addresses containing the first byte of an 8086 opcode. If you set more than ten breakpoints, **debug** returns the **BP** error message.

The user stack pointer must be valid and must have 6 bytes available for this command. The **G** command uses an **IRET** instruction to cause a jump to the program under test. The user stack pointer is set, and the user flags, code segment register, and instruction pointer are pushed on the user stack. (If the user stack is not valid or is too small, the operating system may crash.) An interrupt code (0CCH) is placed at the specified breakpoint address(es).

When **debug** encounters an instruction with the breakpoint code, it restores all breakpoint addresses to their original instructions. If you don't halt execution at one of the breakpoints, the interrupt codes are not replaced with the original instructions.

Using the Go command

Example:

Suppose you type the following command:

```
GCS:7550
```

The program currently in memory would execute up to the address 7550 in the CS segment. **Debug** would then display registers and flags, after which the **G** command would terminate.

After **debug** has encountered a breakpoint, if you type the **G** command again the program runs as if you had typed the filename at the MS-DOS command level. The only difference is that program execution begins at the instruction after the breakpoint, rather than at the usual start address.

Hex

Purpose:

Performs hexadecimal arithmetic on the two specified parameters.

Syntax:

H*value value*

Comments:

First, **debug** adds the two parameters, then subtracts the second parameter from the first. The results of these actions are displayed on one line — first the sum, then the difference.

Example:

Suppose you type the following command:

```
H19F 10A
```

In response, **debug** would perform the calculations and then display the following result:

```
02A9 0095
```


(H)ex

Using the Hex command

(I)nput

Using the Input command

Input

Purpose:

Inputs and displays one byte from the port specified by *value*.

Syntax:

I*value*

Comment:

A 16-bit port address is allowed.

Example:

Suppose you type the following command:

```
I2F8
```

Suppose also that the byte at the port is 42H. **Debug** would input the byte and then display the following:

```
42
```

Load

Purpose:

Loads a file into memory.

Syntax:

L[*address* [*drive*: *record record*]]

Comments:

Set BX:CX to the number of bytes read. The file must have been named either when you started **debug** or with the N (name) command. Both the **debug** invocation and the N command format a filename properly in the normal format of a file control block at CS:5C.

If you use the L (load) command without any parameters, **debug** loads the file into memory beginning at address CS:100 and sets **BX:CX** to the number of bytes loaded. If you type the L command with an address parameter, loading begins at the memory location specified by the *address*. If you use the L command with all parameters included, absolute disk sectors are loaded, instead of a file.

Each *record* is taken from the specified *drive*: (the drive name is numeric: 0 = A:, 1 = B:, 2 = C:, etc.). **Debug** begins loading with the first specified *record*, and continues until the number of sectors in the second *record* have been loaded.

Example:

Suppose, once you have started **debug**, that you type the following commands:

```
-NFILE.COM
```

Now, to load *file.com*, you would simply type the L command.

Debug would then load the file and display the **debug** prompt. Suppose now that you want to load only portions of a file or certain records from a disk. To do this, you would type the following:

```
L04BA:100 2 0F 6D
```

Debug would then load 109 (6DH) records, beginning with logical record number 15, into memory beginning at address 04BA:0100. Then once it had loaded the records, **debug** would simply return the hyphen (-) prompt.

(L)oad

Using the Load command

If the file has an *.exe* extension, it would be relocated to the load address specified in the header of the *.exe* file. The *address* parameter is always ignored for *.exe* files. The header itself is stripped off the *.exe* file before it is loaded into memory. So, the size of an *.exe* file on disk will differ from its size in memory.

If the file was named by the N (name) command, or specified when you started **debug**, as a *.hex* file, then typing the L command with no parameters would cause **debug** to load the file beginning at the address specified in the *.hex* file. If the L command included the option *address*, **debug** would add the *address* specified in the L command to the address found in the *.hex* file to determine the start address at which to load the file.

Move

Purpose:

Moves the block of memory specified by *range* to the location beginning at the specified *address*.

Syntax:

Mrange address

Comments:

Overlapping moves (i.e., moves where part of the block overlaps some of the current addresses) are always performed without loss of data. Addresses that could be overwritten are moved first. For moves from higher to lower addresses, the sequence of events is to first move the data beginning at the block's lowest address and then work toward the highest. For moves from lower to higher addresses, the sequence is to first move the data beginning at the block's highest address and then work toward the lowest.

Note that if the addresses in the block being moved will not have new data written to them, the data in the block *before the move* will remain. The **M** (move) command copies the data from one area into another, in the sequence described, and writes over the new addresses. This action is why the sequence of the move is important.

Example:

Suppose you type the following command:

```
MCS:100 110 CS:500
```

In response, **debug** would first move address CS:110 to CS:510, then CS:10F to CS:50F, and so on until it has moved CS:100 to CS:500. To review the results of the move, you could type the **D** (dump) command, using the same *address* you used with the **M** command.

Using the Move command

Name

Purpose:

Sets filenames.

Syntax:

N*filename* [*filename...*]

Comments:

The **N** (name) command performs two functions. First, it assigns a filename for a later **L** (load) or **W** (write) command. So if you start **debug** without naming a file to be debugged, you must type the command N*filename* before a file can be loaded. Second, the **N** command assigns filename parameters to the file being debugged. In this case, **N** accepts a list of parameters used by the file being debugged.

Note that these two functions overlap. Consider, for example, the following set of **debug** commands:

```
-NFILE1.EXE
-L
-G
```

The **N** command would use these commands to perform the following steps:

1. It would first assign the filename *file1.exe* to the file to be used in any later **L** or **W** commands.
2. It would also assign the *file1.exe* filename to the first filename parameter used by any program that is later debugged.
3. The **L** command would then load *file1.exe* into memory.
4. The **G** (go) command would cause *file1.exe* to be run with *file1.exe* as the single filename parameter (that is, *file1.exe* would be run as if *file1.exe* had been typed at the command level).

A more useful chain of commands might look like this:

```
-NFILE1.EXE
-L
-NFILE2.DAT FILE3.DAT
-G
```

In this example, the N command sets *file1.exe* as the filename for the subsequent L command, which loads *file1.exe* into memory. The N command is then used again, this time to specify the parameters to be used by *file1.exe*. Finally, when the G command is run, *file1.exe* is executed as if *file1 file2.dat file3.dat* had been typed at the MS-DOS command level.

Note that if you were to execute a W command now, then *file1.exe* — the file being debugged — would be saved with the name *file2.dat*. To avoid this kind of result, you should always execute an N command before either an L or W command.

There are four regions of memory that can be affected by the N command:

Memory location	Contents
CS:5C	FCB for file 1
CS:6C	FCB for file 2
CS:80	Count of characters
CS:81	All characters typed

The first filename parameter that you specify for the N command has a file control block (FCB) set up at CS:5C. If you name a second filename parameter, an FCB is set up for this parameter beginning at CS:6C. The number of characters typed in the N command (exclusive of the first character, N) is given at location CS:80.

The actual stream of characters given by the N command (again, exclusive of the letter N) begins at CS:81. Note that this stream of characters may contain switches and delimiters that would be legal in any command typed at the MS-DOS command level.

Example:

A typical use of the N command is as follows:

```
 DEBUG PROG.COM
-NPARAM1 PARAM2/C
-G
-
```

In this case, the G command executes the file in memory as if you had typed the following command line:

```
PROG PARAM1 PARAM2/C
```

Testing and debugging therefore reflect a normal run-time environment for *prog.com*.

Using the Name command

(O)utput

Output

Purpose:

Sends the specified *byte* to the output port specified by *value*.

Syntax:

O*value byte*

Comment:

A 16-bit port address is allowed.

Example:

Using the Output command

Suppose you want to **debug** to output the byte value 4F to output port 2F8. To do this, you could simply type the following command:

```
02F8 4F
```

Quit

Purpose:

Terminates the **debug** utility.

Syntax:

Q

Comments:

The Q (quit) command takes no parameters and exits **debug** without saving the file you're currently working with. You are returned to the MS-DOS command level.

Example:

To end the debugging session, simply type the following and press the RETURN key:

```
q
```

Debug terminates, and control returns to the MS-DOS command level.

(Q)uit

Using the Quit command

(R)egister

Register

Purpose:

Displays the contents of one or more CPU registers.

Syntax:

R[*register-name*]

Comments:

If you do not type a *register-name*, the **R** (register) command dumps the register storage area and displays the contents of all registers and flags.

If you do type a *register-name*, the 16-bit value of that register is displayed in hexadecimal, and a colon then appears as a prompt. You can now either type a *value* to change the register, or press the RETURN key if you don't want a change.

Following is a list of the valid *register-names*:

AX	**BP**	**SS**	
BX	**SI**	**CS**	
CX	**DI**	**IP**	(**IP** and PC both refer
DX	**DS**	**PC**	to the instruction pointer.)
SP	**ES**	**F**	

Any other entry for *register-name* results in a **BR** error message.

If you type **F** as the *register-name*, **debug** displays each flag with a two-character alphabetic code. To change any flag, type the opposite two-letter code. The flags are then either set or cleared.

The flags are listed below with their codes for **SET** and **CLEAR**:

Flag name	Set	Clear
Overflow	**OV**	**NV**
Direction	**DN** (Decrement)	**UP** (Increment)
Interrupt	**EI** (Enabled)	**DI** (Disabled)
Sign	**NG** (Negative)	**PL** (Plus)
Zero	**ZR**	**NZ**
Auxiliary Carry	**AC**	**NA**
Parity	**PE** (Even)	**PO** (Odd)
Carry	**CY**	**NC**

Whenever you type the **RF** command, the flags are displayed (in the order shown in the previous table) in a row at the beginning of a line. At the end of the list of flags, **debug** displays a hyphen (-).

You may enter new flag values in any order as alphabetic pairs. You do not have to leave spaces between these values. To exit the **R** command, press the RETURN key. Any flags for which you did not specify new values remain unchanged.

If you type more than one value for a flag, **debug** returns a **DF** error message. If you enter a flag code other than one of those shown in the table above, **debug** returns a **BF** error message. In both cases, the flags up to the error in the list are changed; those flags at and after the error are not.

When you start **debug**, the segment registers are set to the bottom of free memory, the instruction pointer is set to 0100H, all flags are cleared, and the remaining registers are set to zero.

Using the Register command

Examples:

If you type the following command, **debug** displays all registers, flags, and the decoded instruction for the current location:

```
R
```

If the location is CS:11A, for example, the display will look similar to this:

```
AX=0E00 BX=00FF CX=0007 DX=01FF SP=039D BP=0000
SI=005C DI=0000 DS=04BA ES=04BA SS=04BA CS=04BA
IP=011A  NV UP DI NG NZ AC PE NC
04BA:011A  CD21           INT     21
```

If you then type the following command, **debug** will display these flags:

```
RF
NV UP DI NG NZ AC PE NC - _
```

Now, you could type any valid flag designation, in any order, with or without spaces. For example,

```
NV UP DI NG NZ AC PE NC - PLEICY
```

In response, **debug** would display the **debug** prompt. To see the changes, type either the **R** or **RF** command:

```
RF
NV UP EI PL NZ AC PE CY - _
```

Press the RETURN key to leave the flags this way or to specify different flag values.

Search

Purpose:

Searches the specified *range* for the specified *list* of bytes.

Syntax:

Srange list

Comments:

The *list* may contain one or more bytes, each separated by a space or comma. If the *list* contains more than one byte, only the first address of the byte string is returned. If the *list* contains only one byte, all addresses of the byte in the *range* are displayed.

Example:

Suppose you type the following command:

```
SCS:100 110 41
```

Debug would display a response similar to

```
04BA:0104
04BA:010D
-type:
```

(S)earch

Using the Search command

(T)race

Trace

Purpose:

Executes one instruction and displays the contents of all registers, flags, and the decoded instruction.

Syntax:

T[=*address*] [*value*]

Comments:

If you include the =*address* option in the T (trace) command, tracing occurs at the specified =*address*. The *value* option causes **debug** to execute and trace the number of steps specified by *value*.

The T command uses the hardware trace mode of the 8086 or 8088 microprocessor. Consequently, you may also trace instructions stored in ROM (Read Only Memory).

Example:

Suppose you type the following command:

```
T
```

In response, **debug** would return a display of the registers, flags, and decoded instruction for that one instruction. Assuming, for this example, that the current position is 04BA:011A, **debug** might return the following display:

```
AX=0E00 BX=00FF CX=0007 DX=01FF SP=039D BP=0000
SI=005C DI=0000 DS=04BA ES=04BA SS=04BA CS=04BA
IP=011A  NV UP DI NG NZ AC PE NC
04BA:011A  CD21           INT      21
```

If you type the following command, **debug** executes sixteen (10 hex) instructions beginning at 011A in the current segment, and then displays all registers and flags for each instruction as it is executed. The display scrolls away until the last instruction is executed, and then stops. Now you can see the register and flag values for the last few instructions performed:

```
T=011A 10
```

Remember that if you want to study the registers and flags for any instruction (at any time), you can press CONTROL-S to stop the display from scrolling.

Using the Trace command

Unassemble

Purpose:

Disassembles bytes and displays the source statements that correspond to them, with addresses and byte values.

Syntax:

U[*range*]

Comments:

The display of disassembled code looks like a listing for an assembled file. If you type the U (unassemble) command without parameters, 20 hexadecimal bytes are disassembled at the first address after that displayed by the previous U command. If you type the U command and include the *range* parameter, **debug** disassembles all bytes in *range*. But if you specify *range* only as an *address*, 20H bytes are disassembled.

Example:

Suppose you type the following command:

```
U04BA:100 L10
```

In response, **debug** would disassemble 16 bytes, beginning at address 04BA:0100:

```
04BA:0100  206472      AND   [SI+72],AH
04BA:0103  69          DB    69
04BA:0104  7665        JBE   016B
04BA:0106  207370      AND   [BP+DI+70],DH
04BA:0109  65          DB    65
04BA:010A  63          DB    63
04BA:010B  69          DB    69
04BA:010C  66          DB    66
04BA:010D  69          DB    69
04BA:010E  63          DB    63
04BA:010F  61          DB    61
```


Using the Unassemble command

Now, suppose you type the following:

```
U04BA:0100 0108
```

The display would now show:

```
04BA:0100  206472      AND   [SI+72],AH
04BA:0103  69          DB    69
04BA:0104  7665        JBE   016B
04BA:0106  207370      AND   [BP+DI+70],DH
```

If the bytes in some addresses are altered, the disassembler alters the instruction statements. You can then type the U command for the changed locations, for the new instructions viewed, and for the disassembled code used to edit the source file.

Write

(W)rite

Purpose:

Writes the file being debugged to a disk file.

Syntax:

W[*address* [*drive*: *record record*]]

Comments:

If you do not use parameters with the **W** (write) command, **BX:CX** must already be set to the number of bytes to be written; the file is written beginning from CS:100. If you type the **W** command with just an *address*, the file is written beginning at that *address*. If you have used a **G** (go) or **T** (trace) command, you must reset **BX:CX** before using the **W** command without parameters.

Note that if a file is loaded and modified, the name, length, and starting address are all set correctly to save the modified file (as long as the length has not changed).

You must have named the file either with the initial **debug** startup command or with the **N** (name) command (refer to the **N** command earlier in this manual). Both the **debug** startup command and the **N** command properly format a filename in the normal format of a file control block at CS:5C.

If you include parameters when you use the **W** command, the write begins from the memory address specified; the file is written to the specified *drive*: (the drive name is numeric: 0 = A:, 1 = B:, 2 = C:, etc.). **Debug** writes the file beginning at the logical record number specified by the first *record*. **Debug** then continues to write the file until the number of sectors specified in the second *record* have been written.

Warning Writing to absolute sectors is *extremely risky* because the process bypasses the file handler.

Examples:

If you type the following command, **debug** writes out the contents of memory to the disk in drive B, beginning with the address CS:100. The data written out starts in the disk logical record number 37H and consists of 2BH records.

Using the Write command

```
WCS:100 1 37 2B
```

When the write is complete, **debug** displays its prompt again.

Debug error codes

Debug Error Messages

During a **debug** session, you may receive any of the following error messages. Each error ends the **debug** command under which it occurred, but does not end **debug** itself.

Error code	Definition
BF	Bad flag You attempted to change a flag, but the characters you typed were not one of the acceptable pairs of flag values. See the **R** (register) command for the list of acceptable flag entries.
BP	Too many breakpoints You specified more than ten breakpoints as parameters to the **G** (go) command. Retype the **G** command using ten or fewer breakpoints.
BR	Bad register While using the **R** command you typed an invalid register name. See the **R** command for the list of valid register names.
DF	Double flag You typed two values for one flag. You may specify a flag value only once per **RF** command.

Abel, Peter IBM PC Assembler Language and Programming
Prentice-Hall 1987, pp.xiii + 462

ISBN 0-13-448143-7

Reprint pp.19-36, 443

CHAPTER 2

Machine Execution

Objective: To introduce machine language and the entering and execution of a program in memory.

INTRODUCTION

The basis of this chapter is the use of a DOS program named DEBUG that allows you to view memory, to enter programs, and to trace their execution. The chapter describes how you can enter these programs directly into memory in the code segment area and provides an explanation of each execution step.

The initial exercise has you inspect the contents of particular locations in memory. The first program example uses "immediate" data defined within the instructions for loading into registers and performing arithmetic. The second program example uses data defined separately in the data segment. Tracing these instructions through machine execution provides insight into the operation of a computer and the role of the registers.

You can start right in with no prior knowledge of assembler language or even of programming. All you need is an IBM PC or equivalent micro and a disk containing the DOS operating system.

GETTING STARTED

On startup, diskette users have to insert the DOS diskette in drive A (on the left). If the power is off, turn it on; if the power is already on, press and hold down Ctrl and Alt together, and press Del.

When the working portion of DOS loads into memory, it responds with a *prompt* that asks for the date and time, and then indicates the default disk drive, normally A for diskette and C for hard disk. If this is not the case, change the default by pressing the drive letter, a colon, and the Return key. This is the normal procedure whenever you enter an exercise from this book.

Copied pursuant to S53B of
the Copyright Act as amended
on behalf of Murdoch Uni.
on 17.2.89

VIEWING MEMORY LOCATIONS

In this first exercise, you use the DOS DEBUG program to view the contents of selected memory locations. To initiate the program, key in DEBUG and press Return. DEBUG should load from disk into memory. When fully loaded, DEBUG's prompt, hyphen (-), appears on the screen, and DEBUG is ready to accept your commands. The only command with which this exercise is concerned is D, for dump.

1. *Memory Size.* The first step is to examine the amount of memory that DOS thinks you have installed. Depending on computer model, the value is based on switches set internally, and may indicate less memory than is actually installed. The value is in memory locations hex 413 and 414, which you can view from DEBUG by means of a two-part address:

   - 400 is the segment address, that you type as 40 (the last zero is assumed), and
   - 13 is the offset from the segment address. Key in the following exactly as you see it:

   D 40:13 [and press Return]

   The first two bytes displayed should be memory size in kilobytes and in hexadecimal, with the bytes in reverse sequence. Some examples showing hex reversed, hex corrected, and the decimal equivalent follow:

Hex Reversed	*Hex Corrected*	*Decimal(K)*
80 00	00 80	128
00 01	01 00	256
80 01	01 80	384
00 02	02 00	512
80 02	02 80	640

2. *Serial Number.* A serial number for your computer is embedded in ROM at location hex FE000. To see it, type in:

   D FE00:0 [and press Return]

   A 7-digit number followed by a copyright date should display on the screen.

3. *ROM BIOS Date.* The date of your ROM BIOS begins at hex location FFFF5, recorded as mm/dd/yy. Type

   D FFFF:05 [and press Return]

   Knowing this date is sometimes useful to determine a computer's age and model.

Now that you know how to use the display operation, you can set an address to any valid location to view the contents. You can also step through memory simply by pressing D repeatedly—DEBUG displays locations continuing from the last operation.

When you've completed your inspection, press Q (for quit) and the Return key to exit from DEBUG. Let's now use DEBUG to enter programs directly into memory and trace their execution.

MACHINE LANGUAGE EXAMPLE: IMMEDIATE DATA

The purpose of this example is to illustrate a simple machine language program, how it would appear in main storage, and the effect of its execution. The following program steps are in hexadecimal format:

Instruction	*Explanation*
B82301	Move the hex value 0123 to the AX.
052500	Add the hex value 0025 to the AX.
8BD8	Move the contents of the AX to the BX.
03D8	Add the contents of AX to BX.
8BCB	Move the contents of BX to CX.
2BC8	Subtract the contents of AX from CX.
2BC0	Subtract the contents of AX from AX (clear AX).
90	No operation (do nothing).
CB	Return to DOS.

You may have noticed that machine instructions vary in length: one, two, or three bytes. Also, machine instructions appear in memory one immediately following the other. Program execution begins with the first instruction and steps through each instruction one after another. Do not at this point expect to make much sense of the machine code. For example, in one case a MOV is hex B8 and in another case the MOV is hex 8B.

You can enter this program directly into memory and execute it one instruction at a time. At the same time, you can view the contents of the registers after each instruction. Begin this exercise just as you did for the previous one—key in the command DEBUG and press Return. When DEBUG has fully loaded, its prompt, a hyphen (-) displays. To print this exercise, turn on your printer and press Ctrl and PrtSc together.

To enter the machine language program directly, key in the following command, including the blanks where indicated:

```
E CS:100 B8 23 01 05 25 00  [press Return]
```

The command E stands for Enter. CS:100 indicates the memory address where the instructions are to be entered—hex 100 (256) bytes following the start of the code segment (the normal starting address for machine code under DEBUG). The E command causes each pair of hexadecimal digits to enter a byte in memory, from CS:100 through CS:105.

The next Enter command is

```
E CS:106 8B D8 03 D8 8B CB  [Return]
```

which enters six bytes starting at CS:106 through 107, 108, 109, 10A, and 10B. The last Enter command

```
E CS:10C 2B C8 2B C0 90 CB  [Return]
```

enters six bytes starting at CS:10C through 10D, 10E, 10F, 110, and 111. Double-check that you have keyed in the correct values. If not, you can simply repeat any command that is incorrect.

Now it's just a simple matter of executing these instructions. Figure 2-1 shows all the steps including the E commands. Your screen should display similar results as you enter each DEBUG command.

Key in the command R, followed by Return in order to view the contents of the registers and flags. At this moment DEBUG shows the contents of the registers in hexadecimal format, for example as

AX=0000, BX=0000, ...

Because of differences in DOS versions, the register contents on your screen may not be identical to those shown in Figure 2-1. The IP (instruction pointer) register displays IP=0100 indicating that instruction execution is to begin 100 bytes past the start of the code segment. (That is why you used E CS:100 to enter the start of the program.)

The flags register in Figure 2-1 shows the following flag setting:

NV UP DI PL NZ NA PO NC


```
D>DEBUG
-E CS:100 B8 23 01 05 25 00
-E CS:106 8B D8 03 D8 8B CB
-E CS:10C 2B C8 2B C0 90 CB
-R
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=13C6  ES=13C6  SS=13C6  CS=13C6  IP=0100   NV UP EI PL NZ NA PO NC
13C6:0100 B82301        MOV     AX,0123
-T

AX=0123  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=13C6  ES=13C6  SS=13C6  CS=13C6  IP=0103   NV UP EI PL NZ NA PO NC
13C6:0103 052500        ADD     AX,0025
-T

AX=0148  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=13C6  ES=13C6  SS=13C6  CS=13C6  IP=0106   NV UP EI PL NZ NA PE NC
13C6:0106 8BD8          MOV     BX,AX
-T

AX=0148  BX=0148  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=13C6  ES=13C6  SS=13C6  CS=13C6  IP=0108   NV UP EI PL NZ NA PE NC
13C6:0108 03D8          ADD     BX,AX
-T

AX=0148  BX=0290  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=13C6  ES=13C6  SS=13C6  CS=13C6  IP=010A   NV UP EI PL NZ AC PE NC
13C6:010A 8BCB          MOV     CX,BX
-T

AX=0148  BX=0290  CX=0290  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=13C6  ES=13C6  SS=13C6  CS=13C6  IP=010C   NV UP EI PL NZ AC PE NC
13C6:010C 2BC8          SUB     CX,AX
-T

AX=0148  BX=0290  CX=0148  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=13C6  ES=13C6  SS=13C6  CS=13C6  IP=010E   NV UP EI PL NZ AC PE NC
13C6:010E 2BC0          SUB     AX,AX
-T

AX=0000  BX=0290  CX=0148  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=13C6  ES=13C6  SS=13C6  CS=13C6  IP=0110   NV UP EI PL ZR NA PE NC
13C6:0110 90            NOP
-T

AX=0000  BX=0290  CX=0148  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=13C6  ES=13C6  SS=13C6  CS=13C6  IP=0111   NV UP EI PL ZR NA PE NC
13C6:0111 CB            RETF
-
```

Figure 2-1 Trace of Machine Instructions.

These settings mean, respectively, no overflow, up (right) direction, disable interrupt, plus sign, nonzero, no auxiliary carry, parity odd, and no carry. At this time, none of these flags is important.

The R command also displays at offset 0100 the first instruction to be executed. Note that in Figure 2-1 the CS register contains CS=13C6 (yours may differ) and the instruction is the following:

```
13C6:0100  B82301    MOV AX,0123
```

- CS=13C6 means that the start of the code segment is at location 13C6, or rather, 13C60. The value 13C6:0100 means 100 (hex) bytes following the CS address 13C6.
- B82301 is the machine address that you entered at CS:100.
- MOV AX,0123 is the symbolic assembler instruction for the machine code. This is the result of an "unassembly" operation, which DEBUG displays so that you may more easily interpret the machine instruction. In later chapters, you code assembler instructions exclusively. This assembler instruction means, in effect, Move the immediate value 0123 into the AX register.

At this point, the MOV instruction has not executed. For this purpose, key in T (for trace) and press Return. DEBUG now displays the contents of the registers, the flags, and the next instruction that is to execute. Note that the AX register now contains 0123. The machine code was B8 (Move to AX register) followed by 2301. The operation moves the 23 to the low half (AL) of the AX register and the 01 to the high half (AH) of the AX register:

```
AX: |01|23|
```

The contents of the IP register is 0103 to indicate the location in the code segment of the next instruction to be executed:

```
13C6:0103  052500    ADD  AX,0025
```

To execute this instruction, enter another T. The instruction adds 25 to the low half (AL) of the AX register and 00 to the high half (AH) of the AX, in effect adding 0025 to the AX. AX now contains 0148, and IP contains 0106 for the next instruction to be executed.

Key in another T command. The instruction moves the contents of the AX register to the BX register—note that BX now contains 0148. AX still contains 0148 too because the MOV operation actually copies the data from one location to another.

Now key in a T command to step through each remaining instruction in the program. The next instruction adds the contents of AX to BX, giving 0290 in BX. Then the program moves (copies) the contents of BX into CX, subtracts AX from CX, and subtracts AX from itself. After this last operation, the zero flag is changed from NZ (nonzero) to ZR (zero) to indicate that the result of the last operation was zero (subtracting AX from itself cleared it to zero).

Although you can also press T for the last instructions, NOP and RET, it is not recommended going past this point. To view the machine language program in the code segment, key in D for Dump as follows:

```
D CS:100
```

On an 80-column screen, DEBUG displays 16 bytes (32 hex digits) to the left of each line. To the right is the ASCII representation (if printable) of each byte (pair of hex digits). In the case of machine code, the ASCII representation is meaningless and may be ignored. Later sections discuss the right side of the dump in more detail.

The first line of the dump begins at 00 and represents the contents of locations CS:100 through CS:10F. The second line represents the contents of CS:110 through CS:11F. Although your program ends at CS:111, the Dump command automatically displays eight lines (on an 80-column screen) from CS:100 through CS:170.

If you want to reexecute these instructions, reset the IP register and trace through again. Enter R IP, enter 100, then R, and the required number of T commands, all followed by Return.

Figure 2-2 shows the results of the D CS:100 command. Expect only the machine code from CS:100 through 111 to be identical to your own dump; the bytes that follow could contain anything.

```
-D CS:100
13C6:0100  B8 23 01 05 25 00 8B D8-03 D8 8B CB 2B C8 2B C0   .#..%.......+.+.
13C6:0110  90 CB 8D 46 14 50 51 52-FF 76 28 E8 74 00 8B E5   ...F.PQR.v(.t...
13C6:0120  B8 01 00 50 FF 76 32 FF-76 30 FF 76 2E FF 76 28   ...P.v2.v0.v..v(
13C6:0130  E8 88 15 8B E5 FF 36 18-12 FF 36 16 12 8B 76 28   ......6...6...v(
13C6:0140  FF 74 3A 89 46 06 E8 22-CE 8B E5 30 E4 3D 0A 00   .t:.F.."...0.=..
13C6:0150  75 32 A1 16 12 2D 01 00-8B 1E 18 12 83 DB 00 53   u2...-.........S
13C6:0160  50 8B 76 28 FF 74 3A A3-16 12 89 1E 18 12 E8 FA   P.v(.t:.........
13C6:0170  CD 8B E5 30 E4 3D 0D 00-74 0A 83 06 16 12 01 83   ...0.=..t.......
-Q
```

Figure 2-2 Dump of the Code Segment.

To terminate the DEBUG session, enter Q (for Quit). This operation returns you to DOS, which displays its prompt A> or C>. If you printed the session, press Ctrl/PrtSc again to terminate printing.

MACHINE LANGUAGE EXAMPLE: DEFINED DATA

The previous example used immediate values defined directly within the first two instructions (MOV and ADD). We next illustrate a similar example that defines the two data fields 0123 and 0025 in the data segment. Working through this example should provide insight into how a computer accesses data by means of the DS register and offset addresses.

The present example defines data items containing, respectively, the following:

DS Location	*Hex Value*	*Bytes Occupied*
0000	2301	0 and 1
0002	2500	2 and 3
0004	0000	4 and 5
0006	2A2A2A	6, 7, and 8

Remember that a hex character occupies a half-byte, so that, for example, 23 is stored in byte 0 (the first byte) of the data segment, and 01 is stored in byte 1 (the second byte).

The machine language instructions in the code segment that process these items are the following:

Instruction	*Explanation*
A10000	Move the word (two bytes) beginning at DS location 0000 into the AX register.
03060200	Add the contents of the word (two bytes) beginning at DS location 0002 to the AX register.
A30400	Move the contents of the AX register to the word beginning at DS location 0004.
CB	Return to DOS.

You may have noticed that there are two MOV instructions with different machine codes: A1 and A3. The actual machine code is dependent on the registers that are referenced, the number of bytes (byte or word), the direction of data transfer (from or to a register), and the reference to immediate data or memory.

Once again, you can use DEBUG to enter this program and to watch its execution. When DEBUG signals its hyphen prompt, it is ready for your commands.

First key in the Enter (E) commands for the data segment:

```
E DS:00 23 01 25 00 00 00  [press Return]
E DS:06 2A 2A 2A  [press Return]
```

The first command stores the three words (six bytes) at the start of the data segment, DS:00. Note that you have to enter words with the bytes reversed so that 0123 is 2301 and 0025 is 2500. When a MOV instruction accesses these words into a register, it "unreverses" the bytes so that 2301 becomes 0123 and 2500 becomes 0025.

The second command stores three asterisks (***) so that you can view them later using the Dump (D) command—the asterisks serve no purpose in the data segment. Now key in the instructions in the code segment, once again beginning at CS:100, as follows:

```
E CS:100 A1 00 00 03 06 02 00
E CS:107 A3 04 00 CB
```

These instructions are now found in memory locations CS:100 through CS:10A. You can execute them just as you did earlier. Figure 2-3 shows all the steps including the E commands. Your screen should display similar results, although the CS and DS addresses probably differ. You may want to look at the stored data in the data segment and the instructions in the code segment. Key in the Dump commands respectively:

To view the data segment: `D DS:000 [press Return]`
To view the code segment: `D CS:100 [press Return]`

Compare the contents of both segments to what you keyed in and to Figure 2-3. DS:00 through 08 and CS:100 through 10A should be identical to Figure 2-3.

Now press R to view the contents of the registers and flags and to display the first instruction. The registers contain the same values as at the start of the first example. The displayed instruction is

```
13C6:0100  A10000    MOV AX,[0000]
```

Since the CS register contains 13C6, then CS:0100 contains your first instruction, A10000. DEBUG interprets this instruction as a MOV and has determined that the reference is to the first location [0000] in the data segment. The square brackets are to tell you that this reference is to a

```
>DEBUG
-E DS:000 23 01 25 00 00
-E DS:006 2A 2A 2A
-E CS:100 A1 00 00 03 06 02 00
-E CS:107 A3 04 00 CB
-D DS:0
13C6:0000  23 01 25 00 00 9A 2A 2A-2A F0 F5 02 2C 10 2E 03   #.%...***...,...
13C6:0010  2C 10 BD 02 2C 10 B1 0D-01 03 01 00 02 FF FF FF   ,...,...........
13C6:0020  FF FF FF FF FF FF FF FF-FF FF FF FF EF 0F 64 00   ..............d.
13C6:0030  61 13 14 00 18 00 C7 13-FF FF FF FF 00 00 00 00   a...............
13C6:0040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
13C6:0050  CD 21 CB 00 00 00 00 00-00 00 00 00 00 20 20 20   .!...........
13C6:0060  20 20 20 20 20 20 20 20-20 00 00 00 00 00 20 20 20          .....
13C6:0070  20 20 20 20 20 20 20 20-20 00 00 00 00 00 00 00 00        ........
-R
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=13C6  ES=13C6  SS=13C6  CS=13C6  IP=0100   NV UP EI PL NZ NA PO NC
13C6:0100 A10000        MOV     AX,[0000]                       DS:0000=0123
-T

AX=0123  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=13C6  ES=13C6  SS=13C6  CS=13C6  IP=0103   NV UP EI PL NZ NA PO NC
13C6:0103 03060200      ADD     AX,[0002]                       DS:0002=0025
-T

AX=0148  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=13C6  ES=13C6  SS=13C6  CS=13C6  IP=0107   NV UP EI PL NZ NA PE NC
13C6:0107 A30400        MOV     [0004],AX                       DS:0004=9A00
-T

AX=0148  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=13C6  ES=13C6  SS=13C6  CS=13C6  IP=010A   NV UP EI PL NZ NA PE NC
13C6:010A CB            RETF
-D DS:0
13C6:0000  23 01 25 00 48 01 2A 2A-2A F0 F5 02 2C 10 2E 03   #.%.H.***...,...
13C6:0010  2C 10 BD 02 2C 10 B1 0D-01 03 01 00 02 FF FF FF   ,...,...........
13C6:0020  FF FF FF FF FF FF FF FF-FF FF FF FF EF 0F 64 00   ..............d.
13C6:0030  61 13 14 00 18 00 C7 13-FF FF FF FF 00 00 00 00   a...............
13C6:0040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
13C6:0050  CD 21 CB 00 00 00 00 00-00 00 00 00 00 20 20 20   .!...........
13C6:0060  20 20 20 20 20 20 20 20-20 00 00 00 00 00 20 20 20          .....
13C6:0070  20 20 20 20 20 20 20 20-20 00 00 00 00 00 00 00 00        ........
-Q
```

Figure 2-3 Trace of Machine Instructions.

memory address and not an immediate value. An immediate value that would move zeros to the AX register would appear as

```
MOV  AX,0000
```

Now key in the DEBUG command T. The instruction MOV AX,[0000] moves the contents of the word at offset zero in the data segment to the AX register. The contents are 2301, which the operation reverses in AX as 0123.

The next instruction is ADD, which you can cause to execute by keying in another T command. The operation adds the contents of the word at DS offset 0002 to the AX register. The result in the AX is now the sum of 0123 and 0025, or 0148.

The next instruction is MOV [0004],AX. Key in a T command for it to execute. The instruction moves the contents of the AX register to the word at DS offset 0004. To view the changed contents of the data segment, key in D DS:00. The first nine bytes are

Value in data segment:	23 01 25 00 48 01 2A 2A 2A
Offset number:	00 01 02 03 04 05 06 07 08

The value of 0148 that was moved from the AX register to offset 04 and 05 is reversed as 4801. Note that these hex values are represented on the right of the screen by their ASCII equivalents. For example, hex 23 generates a number (#) symbol and hex 25 generates a percent (%) symbol. The three hex 2A bytes generate asterisks (*). The left side of the dump shows the actual machine code as it appears in memory. The right side of a dump simply helps you locate alphabetic (string) data more easily.

To view the contents of the code segment, key in D CS:100, just as Figure 2-3 shows. Finally, enter Q (for Quit) to terminate the DEBUG session.

MACHINE ADDRESSING

To access an instruction, the processor determines its address from the contents of the CS register plus the offset in the IP register. For example, assume that the CS contains hex 04AF (actually 04AF0) and the IP contains hex 0023:

```
CS:                 04AF0
IP:                  0023
Instruction address: 04B13
```

Let's say that the instruction beginning at 04B13 is the following:

```
A11200          MOV  AX,[0012]
|
location 04B13
```

Memory location 04B13 contains the first byte of the instruction to be accessed. The processor determines from the operation itself (A1) that the instruction is three bytes long.

To access the data item at offset [0012], the processor determines its location from the contents of the DS register (usually) plus the offset in the instruction operand. If the DS contains hex 04B1 (actually 04B10), then the actual location of the referenced data item is

```
DS:              04B10
Offset:        + 0012
Address of data: 04B22
```

Let's say that the contents of locations 04B22 and 04B23 are the following:

```
Contents:    24   01
             |    |
Location: 04B22   04B23
```

The processor extracts the 24 at location 04B22 and inserts it into the AL register, and the 01 at location 04B23 into the AH register. The AX now contains 0124. As the processor fetches each byte of the instruction, it increments the IP register so that it contains the offset (0026) for the next instruction. The processor is now ready to execute the next instruction, which it derives once again from the contents of the CS (04AF0) plus the current offset in the IP (0026), in effect 04B16.

Even-Numbered Addresses

The 8086, 80286 and 80386 processors operate more efficiently if an accessed word begins on an even-numbered address. In the previous example, the processor can access the word at 4B22 directly into a register. But if the word begins on an odd-numbered address, the processor performs two accesses. For example, assume that an instruction has to access a word beginning at location 04B23 into the AX register:

```
Memory contents: |xx|24|01|xx|
                      |
Location:        04B23
```

The processor first accesses the bytes at 4B22 and 4B23 and delivers the byte from 4B23 to the AL register. It then accesses the bytes at 4B24 and 4B25 and delivers the byte from 4B24 to the AH register. The AX now contains 0124.

You don't have to perform any special programming for even or odd locations, nor do you have to know whether an address is odd. The

significant points are (1) the accessing operation reverses a word from memory into a register so that it resumes its correct sequence, and (2) if you have a routine that accesses memory repetitively, for efficiency you could define the data to begin at an even location.

For example, since the beginning of the data segment is always an even address, the first data field begins on an even number, and as long as successive data fields are defined as even-numbered words, they all begin on even addresses. For most purposes, however, these processors execute at such rapid speed that you'll never notice the effects of this efficiency.

The assembler has an EVEN pseudo-operation that aligns data and instructions on even memory locations.

MACHINE LANGUAGE EXAMPLE: MEMORY SIZE DETERMINATION

In the first exercise in this chapter, you checked the amount of memory (RAM) that your computer contains. BIOS (basic input/output system) in ROM has a routine that delivers memory size. You can access BIOS through INT instructions, in this case an interrupt 12H. BIOS then returns the value to the AX register in terms of 1K bytes. Load DEBUG into memory and enter the following machine code for INT 12H and RET:

```
E CS:100 CD 12 CB
```

Press R (and Return) to display the registers and the first instruction. The AX contains 0000 and the IP contains 0100; the instruction is INT 12H. Now press T (and Return) repeatedly and see the following BIOS instructions execute (DEBUG shows the symbolic instructions, although it is the machine code that actually executes):

```
STI
PUSH  DS
MOV   AX,0040
MOV   DS,AX
MOV   AX,[0013]
POP   DS
IRET
```

At this point, the AX contains the size of memory in hexadecimal format. Now enter another T command to exit from BIOS and to return to your program. The displayed instruction is RET for the machine code CB that you entered.

SPECIAL DEBUG FEATURES

Under DOS 2.0 and on, you can use DEBUG to enter assembler instructions as well as machine language instructions. You'll find occasions to use both methods.

The A Command

The DEBUG A (Assemble) command tells DEBUG to begin accepting assembler instructions and to convert them into machine language. Initialize the starting address as

```
A 100  [Return]
```

DEBUG displays the value of the code segment and the offset, as xxxx:0100. Type in each instruction followed by the Return key. When you've entered the program, press Return again to exit the A command. Try entering the following program:

```
MOV  AL,25    [Return]
MOV  BL,32    [Return]
ADD  AL,BL    [Return]
RET           [Return]
```

On completion, the screen should display the following:

```
xxxx:0100    MOV AL,25
xxxx:0102    MOV BL,32
xxxx:0104    ADD AL,BL
xxxx:0106    RET
```

At this point, DEBUG is ready to accept another instruction. Just press Return again to terminate the operation.

You can see that DEBUG has determined the starting location of each instruction. But before executing the program, let's examine the generated machine language.

The U Command

DEBUG's U (Unassemble) command displays the machine code for your assembler instructions. You have to tell DEBUG the locations of the first and last instructions that you want to see, in this case, 100 and 106. Key in


```
U 100,106  [and Return]
```

The screen should display

```
xxxx:0100    B025  MOV  AL,25
xxxx:0102    B332  MOV  BL,32
xxxx:0104    00D8  ADD  AL,BL
xxxx:0106    C3    RET
```

Now trace execution of the program. Begin with R to display the registers and the first instruction, and T to trace subsequent instructions.

You can now see how to enter a program in either machine or assembly language. A common use for entering assembler is where you don't know the machine code; a common use for entering machine code is to change values in a program during execution. However, DEBUG is really intended for what its name implies—debugging programs—and most of your efforts will involve use of the assembler program, as discussed in the next chapter.

Saving a Program from Within DEBUG

You may use DEBUG to save a program on disk under two circumstances.

1. You have read a program, modified it, and now want to save it. Follow these steps:
   - Read the program under its name: DEBUG n:filename [Return]
   - Use the D command to view the program and E to enter changes.
   - Write the revised program: W [Return].
2. You have used DEBUG to write a very small program that you now want to save.
   - Request the DEBUG program.
   - Use A (assemble) and E (enter) to create the program.
   - Name the program: N filename.COM [Return]. The program extension must be COM—see Chapter 6 for details of COM files.
   - Since only you know where the program really ends, tell DEBUG the length of the program in bytes. In the preceding example, the last instruction was

     xxxx:0106 C3 RET

The last instruction is one byte, and therefore the program size is 106 (end) minus 100 (start), or 6.

- First request the CX register as R CX [Return].
- DEBUG replies with CX 0000 (zero value).
- You reply with the program length, 6.
- Write the revised program: W [Return].

For both methods, DEBUG displays a message "Writing nnnn bytes." If the number is zero, you have failed to enter the program length; try again.

KEY POINTS TO REMEMBER

DOS DEBUG is a powerful program that is useful for debugging assembler programs. But be careful in its use, especially the Enter (E) command. Entering data at a wrong location or entering incorrect data may cause unpredictable results. You may find your screen filled up with strange characters, have your keyboard lock, or even cause DOS to interrupt DEBUG and reload itself from disk. You are not likely to cause any damage, but you may get a bit of a surprise and may lose data that you entered during the DEBUG session.

If you enter incorrect values in the data segment or code segment, reenter the E command to correct them. However, you may not have noticed the error until after beginning the trace. You can still use E to make the changes, but you may want to begin execution back at the first instruction. No problem—just set the instruction pointer (IP) register to 0100. Key in the R (register) command followed by the designated register:

```
R IP  [Return]
```

DEBUG displays the contents of the IP and waits for an entry. Key in the value 0100 (followed by Return). To check the result, key in an R command (without the IP). DEBUG displays the registers, flags, and the first instruction to be executed. You can now use T to trace through the instruction steps again.

If your program is accumulating totals, you may have to clear to zero some memory locations and registers. But be sure not to change the contents of the CS, DS, SP, and SS registers, all of which have specific purposes.

Read the chapter in the DOS manual on DEBUG. The parts that are relevant at this time are all the introductory material and the following

DEBUG commands: Dump (D), Enter (E), Hexadecimal (H), Name (N), Quit (Q), Register (R), Trace (T), and Write (W). You may want to examine the other DEBUG commands to see what they do.

QUESTIONS

2-1. Determine the machine instruction for each of the following:

(a) Move the hex value 4629 to the AX register.

(b) Add the hex value 036A to the AX register.

2-2. Assume that you have entered the following E command:

```
E CS:100 B8 45 01 05 25 00
```

The hex value 45 was supposed to be 54. Code an E command to correct only the one byte that is incorrect, that is, directly change 45 to 54.

2-3. Assume that you have entered the following E command:

```
E CS:100 B8 04 30 05 00 30 CB
```

(a) What are the three instructions here? (The first program in this chapter gives a clue.)

(b) On executing this program, you discover that the AX register ends up with 6004 instead of 0460. What is the error and how would you correct it?

(c) Having corrected the instructions, you now want to reexecute the program from the first instruction. What two commands are required?

2-4. The following is a machine language program:

```
B0 25 D0 E0 B3 15 F6 E3 CB
```

The program performs the following:

- Moves the hex value 25 to the AL register.
- Shifts the contents of the AL one bit left (the result is 4A).
- Moves the hex value 15 to the BL register.
- Multiplies AL by BL.

Use DEBUG to enter (E) this program beginning at CS:100. Remember that these are hexadecimal values. After entering the

program, key in D CS:100 to view it. Then key in R and enough successive T commands to step through the program until reaching RET. What is the final product in the AX register?

2-5. Use DEBUG to enter (E) the following machine language program:

```
Data:          25 15 00 00
Machine code:  A0 00 00 D0 E0 F6 26 01 00 A3 02 00 CB
```

The program performs the following:

- Moves the contents of the one byte at DS:00 (25) to the AL register.
- Shifts the AL contents one bit left (the result is 4A).
- Multiplies the AL by the one byte contents at DS:01 (15).
- Moves the product from the AX to the word beginning at DS:02.

After entering the program, key in D commands to view the data segment and code segment. Then key in R and enough successive T commands to step through the program until reaching RET. At this point, the AX should contain the product 0612. Key in another D DS:00 and note that the product at DS:02 is stored as 1206.

2-6. For Question 2-5, code the commands that write the program on disk under the name TRIAL.COM.

2-7. Use the DEBUG A command to enter the following instructions:

```
MOV  BX,25
ADD  BX,30
SHL  BX,01
SUB  BX,22
NOP
RET
```

Unassemble and trace execution through to the NOP.

Answers to Selected Questions

1-1. (a) 01001101; (c) 00111111.

1-2. (a) 0101; (c) 10101.

1-3. (a) 00100010; (c) 00100000.

1-4. (a) 11101101; (c) 11000110.

1-5. (a) 00111000; (c) 10000000.

1-6. (a) 51; (c) 5D.

1-7. (a) 23C8; (c) 8000.

1-8. (a) 13; (c) 59; (e) FFF.

1-9. (a) Stack, data, and code; (c) paragraph.

1-10. (a) AX, BX, CX, DX, DI, SI; (c) AX and DX; (e) flags.

1-11. (a) ROM (read-only memory) is permanent and performs startup procedures and handles input/output. (b) RAM (random/access memory) is temporary and is the area where programs and data reside when executing.

2-1. (a) B82946.

2-2. E CS:101 54.

2-3. (a)
```
MOV AX,3004
ADD AX,3000
RET
```
(c) R and IP to set the IP to 0.

2-4. The product is hex 0612.

2-6. Use the N command to name the program (as COM), set the length in the CX, and use the W command to write it.

3-1. TITLE and PAGE.

3-2. (a), (b), and (c) are valid; (d) is invalid because it starts with a number; (e) is valid only if it refers to the AX register.

3-4. (a) END; (c) ENDS.

3-6. RET is an instruction that causes control to return to the operating system; END is a pseudo-op that tells the assembler that there are no more instructions to assemble.

3-7. `ASSUME CS:CDSEG,DS:DATSEG,SS:STKSEG.`

LABORATORY 6

Processing Data Arrays

Purpose To learn how to process data arrays.

What You Should Learn

1. What identifies elements of an array.
2. The best way to process arrays with the 8088.
3. How to do a summation.
4. How to use a terminator.
5. How to determine whether numbers are within limits.
6. How to display a message.

Reference Materials

Eccles, W. J. *Microprocessor Systems: A 16-Bit Approach*. Reading, MA: Addison-Wesley, 1985, pp. 107–109 (programs with loops), 119–122, 133–137, 187–190, 311–323 (program examples).

Gorsline, G. W. *16-Bit Modern Microcomputers*. Englewood Cliffs, NJ: Prentice-Hall, 1985, pp. 80–88 (comparisons and flow of control), 88–93 (loop control), 106–107 (string data movement).

Liu, Y. C., and G. A. Gibson. *Microcomputer Systems: The 8086/8088 Family* (2nd ed.). Englewood Cliffs, NJ: Prentice-Hall, 1986, pp. 35–37 (addressing modes), 47–50 (instruction execution timing), 65–69 (binary arithmetic), 86–90 (LOOP instruction), 207–212 (string instructions).

Norton, P. *Programmer's Guide to the IBM PC*. Redmond, WA: Microsoft Press, 1985, chap. 9 (ROM-BIOS video services).

Leventhal, Lance A. Microcomputer experimentation with the IBM PC Holt Rinehart and Winston 1988. pp.xviii + 446.
ISBN 0 03 009542 5
Reprint pp99-120, 172-194

Copied pursuant to S53B of the Copyright Act as amended on behalf of Murdoch Uni. on 17.2.89

Terms

Array: set of related data items.

Autodecrement: decreasing a pointer or index automatically while executing an instruction.

Autoincrement: increasing a pointer or index automatically while executing an instruction.

Borrow: bit that is set (1) if the result of a subtraction is negative and cleared (0) if it is positive or zero. Borrows are used to subtract numbers that are too long for a single operation.

Checksum: logical sum used to guard against errors.

Indirect address: address that contains the data's address, as opposed to a direct address that contains the actual data.

Indirect addressing: an addressing method in which the instruction refers to a storage location containing the effective address.

Limit checking: determining if data is within limits, that is, below an upper threshold and above a lower threshold. This procedure can be used to reject invalid data resulting from operator or communications errors. Typical examples of such data are a transaction dated February 30 and a room temperature setting of 70°C (instead of °F).

Logical sum: binary sum with no carries between bit positions.

Pointer: register or memory location that contains an address rather than data.

Register indirect addressing: indirect addressing in which the effective address is in a register.

String: an array of characters (8-bit elements).

String instruction: an instruction that performs one step in a string operation. The step usually involves updating indexes, counters, or pointers as well as doing a basic task. String instructions are also called *string primitives*, as they serve as building blocks for complex string operations.

Terminator: item that marks the end of an array.

8088 Instructions

CLD: clear direction (D) flag; make the direction flag 0, thus causing subsequent string instructions to increment their pointers.

JA (or JNBE): jump if above (not below or equal); jump if carry clear (0) and result not zero (Zero flag = 0).

JBE (or JNA): jump if below or equal (not above); jump if carry set (1) or result is zero (Zero flag = 1).

LODSB: load byte string; load register AL from the location whose address is in register SI, then either increment or decrement SI by 1 depending on whether the D flag is 0 (increment) or 1 (decrement).

LOOP: subtract 1 from register CX and jump to the specified address if CX is not zero.

STD: set direction (D) flag; make the direction flag 1, thus causing subsequent string instructions to decrement their pointers.

Data Arrays

Most computer tasks involve applying instructions to sets of related data, or *arrays*. Typical array operations are calculating averages and other statistics, finding the largest element for scaling, organizing data for storage on tape or disk, editing, sorting, arranging sequences of operations, and searching for commands or entries.

The elements of arrays are usually stored in consecutive memory addresses. Two items are needed to reach an element:

1. The array's starting (*base*) address.
2. The element's number, or *index*.

We often refer mathematically to an element as A_i, where A identifies the entire array (that is, base address), and i identifies the particular element (that is, index).

Flexible addressing modes are the keys to processing arrays. One sequence of instructions should be able to process any element. Otherwise, minor changes in the arrays' locations or lengths will require major revisions in the program. A flexible addressing mode, such as indexed or indirect addressing, allows an instruction to use different effective addresses at different times.

Problem 6-1

Which of these instructions could handle any element of an array? Why?

a. MOV AL,[400H]
b. MOV BX,3E0H
c. MOV AL,[BX+380H]
d. AND AL,[400H]

Which instruction can load data from different memory locations at different times even if the program is in read-only memory?

Problem 6-2

If an array starts at address BASE and each element occupies one byte, where is the second element? Assume that BASE contains the zeroth element. Where is the *j*th element?

Problem 6-3

How do the answers to Problem 6-2 change if each element occupies two bytes? What if each element occupies *k* bytes?

Problem 6-4

We can store a two-dimensional array either by row or by column. For example, we can store an array *A* with *m* rows and *n* columns by row, starting at row 1, column 1. Denoting the element in row *i* and column *j* as A_{ij}, the order in memory is: $A_{11}, A_{12}, A_{13}, \ldots, A_{1n}, A_{21}, A_{22}, A_{23}, \ldots, A_{m1}, A_{m2}, A_{m3}, \ldots, A_{mn}$. If A_{11} is in address B, where is A_{23}? Where is A_{ij}? What is the lowest address occupied by A_{ij} if each element occupies *k* bytes? Note that we are starting the indexes at 1 here, *not* at 0.

Problem 6-5

Assume that an array contains the angles at which a vehicle should move (0 to 359 deg) and the number of minutes (0 to 59) for which it should travel at each angle. Each entry consists of 3 bytes; the first two contain the angle and the third, the travel time at that angle. If the first angle is in addresses BASE and BASE + 1, where will you find:

a. the third angle?
b. the travel time at the fifth angle?
c. the sixth angle?

Processing Arrays with the 8088 Microprocessor

The fastest and simplest way to process arrays with the 8088 is as follows (see Figure 6-1 for a flowchart):

1. Put the array's base address in an index or base register. You can use BX, SI (source index register), or DI (destination index register). These registers serve as *pointers*, as they contain addresses rather than data. We will generally use SI or DI, leaving BX for 8-bit operations. Note that BX is byte-addressable (the bytes are BH and BL), whereas SI and DI are not.

2. Refer indirectly to an element of the array, that is, as [BX], [SI], or [DI]. This is just like indexing, except with no displacement. We call this mode *register indirect addressing.*
3. Move the pointer to the next element with INC SI or INC DI (or to the previous element with DEC SI or DEC DI).
4. Access an arbitrary element by using its offset as a displacement in indexed addressing. For example, ADD AL,[SI+5] will add to AL an element located five bytes ahead of where SI is currently pointing.

The key point here is that only step 1 (initialization) depends on where the array is located.

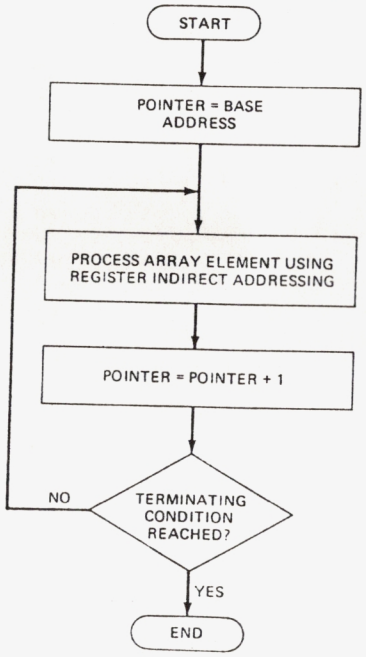


Figure 6-1 Flowchart for array processing with the 8088 microprocessor.

Problem 6-6

Write a program that logically ANDs AL with the memory location three bytes ahead of the one addressed by SI. It should store the result in the location one byte before the address in SI.

EXAMPLE:

[SI] = 0440

RESULT:

[043F] = [0440] AND [0443]

Remember that the brackets mean "contents of."

Note that the 8088 has a short form for signed 8-bit displacements (between −128 and +127). As in conditional jumps, it represents −1 as FF, −2 as FE, and so forth. Positive values (most significant bit = 0) are taken literally.

Sum of Data

A simple example of array processing is summing the elements. This is an essential step in computing averages, variances, or numerical integrals. The following program assumes an array of four elements in 420 through 423 (see Figure 6-2 for a flowchart). The pointer is in register SI.

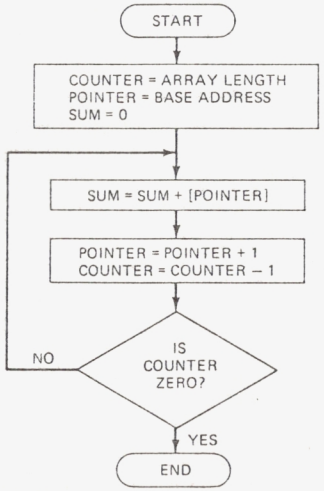


Figure 6-2
Flowchart for summation program.

```
MOV   CX,4          ;COUNT = 4
MOV   SI,420H       ;POINT TO START OF ARRAY
SUB   AL,AL         ;SUM = ZERO INITIALLY
ADDELM: ADD   AL,[SI]     ;ADD ELEMENT TO SUM
        INC   SI
        DEC   CX          ;COUNT ELEMENTS
        JNZ   ADDELM
        MOV   [400H],AL   ;SAVE SUM
        INT   20H
```


PROGRAM 6-1

Memory Address (Hex)	Memory Contents (Hex)		Instruction (DEBUG Form)	
0200	*B9*		*MOV*	*CX,4*
0201	*04*			
0202	*00*			
0203	*BE*		*MOV*	*SI,420*
0204	*20*			
0205	*04*			
0206	*28*		*SUB*	*AL,AL*
0207	*C0*			
0208	*02*	*ADDELM:*	*ADD*	*AL,[SI]*
0209	*04*			
020A	*46*		*INC*	*SI*
020B	*49*		*DEC*	*CX*
020C	*75*		*JNZ*	*208*
020D	*FA*			
020E	*A2*		*MOV*	*[400],AL*
020F	*00*			
0210	*04*			
0211	*CD*		*INT*	*20*
0212	*20*			

Program 6-1 is the DEBUG version. Run it with the following data:

[0420] = 07
[0421] = 23
[0422] = 31
[0423] = 20

RESULT:

[0400] = 7B

Remember that the data is hexadecimal. You can use the command

```
E 420 7 23 31 20
```

to enter it. Change 422 to F1 and run the program again. What is the result, and why?

Problem 6-7

Write a program that sums six elements starting at 420.

SAMPLE PROBLEM:

```
[0420] = 07
[0421] = 23
[0422] = 31
[0423] = 20
[0424] = 16
[0425] = 38
```

RESULT:

```
[0400] = C9
```

Problem 6-8

Make Program 6-1 EXCLUSIVE OR the elements instead of adding them. The result, called a *logical sum* or *checksum*, is often used to detect errors in tape or disk records.

SAMPLE PROBLEM: (four elements, starting at 420, result in 400)

```
[0420] = 07
[0421] = 23
[0422] = 31
[0423] = 20
```

RESULT:

```
[0400] = 35
```


Problem 6-9

Extend Program 6-1 to save the carries and store the 16-bit sum in 400 and 401 (more significant byte in 401).

SAMPLE PROBLEM:

```
[0420] = F7
[0421] = 23
[0422] = 31
[0423] = 20
[0424] = 16
```

RESULT:

```
[0400] = 81 (less significant byte of sum)
[0401] = 01 (more significant byte of sum)
```

The 8088 has a LOOP instruction that replaces the common countdown sequence

```
DEC  CX
JNZ  BEGLOOP
```

LOOP uses CX automatically. For example, in Program 6-1, we can replace the DEC CX, JNZ ADDELM sequence with LOOP ADDELM. The savings are one byte of memory and one clock cycle. Like JNZ, LOOP requires an 8-bit relative offset. The end of the program (locations 20B on) is now

Memory Address (Hex)	Memory Contents (Hex)	Instruction (DEBUG Form)
020B	*E2*	*LOOP 208*
020C	*FB*	
020D	*A2*	*MOV [400],AL*
020E	*00*	
020F	*04*	
0210	*CD*	*INT 20*
0211	*20*	

The D command has a form for displaying arrays of known length. In it, you follow D with:

1. Starting address
2. L
3. Number of bytes.

For example, you can display the data and result from Problem 6-9 with the commands

D 420 L 5 (display five bytes starting at 420)
D 400 L 2 (display two bytes starting at 400)

Note that the L option works with the D command but not with E.

Using a Terminator

If you are not sure how long the array is (or do not want to count the elements), you can follow the data with a special marker, or *terminator*. It must have a value that cannot be a real data item. In a summation, zero is a good choice because it does not affect the sum. The program using zero as a terminator is (see Fig. 6-3 for a flowchart)

```
        MOV   SI,420H      ;POINT TO START OF ARRAY
        SUB   BL,BL        ;SUM = ZERO INITIALLY
ADDELM: MOV   AL,[SI]      ;GET NEXT ELEMENT
        TEST  AL,AL        ;IS IT TERMINATOR (ZERO)?
        JZ    DONE         ;YES, DONE
        ADD   BL,AL        ;NO, ADD ELEMENT TO SUM
        INC   SI
        JMP   ADDELM
DONE:   MOV   [400H],BL    ;SAVE SUM
        INT   20H
```

TEST AL,AL sets the flags from AL's contents. Remember that MOV (as well as IN, OUT, and other data movement instructions) does not affect the flags. ANDing or ORing a register with itself does nothing except set the flags (see Table 3-3). Thus AND reg,reg, OR reg,reg, and TEST reg,reg all set the flags from a register. We chose TEST only to emphasize that the register does not change. A good way to confuse readers is to select among the three at random.

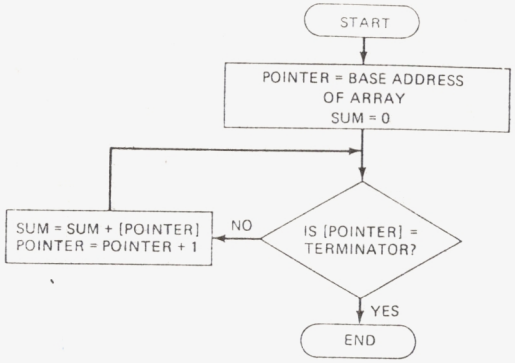


Figure 6-3
Flowchart for summation program with a terminator.

PROGRAM 6-2

Memory Address (Hex)	Memory Contents (Hex)		Instruction (DEBUG Form)
0200	BE		MOV SI,420
0201	20		
0202	04		
0203	28		SUB BL,BL
0204	DB		
0205	8A	ADDELM:	MOV AL,[SI]
0206	04		
0207	84		TEST AL,AL
0208	C0		
0209	74		JZ 210
020A	05		
020B	00		ADD BL,AL
020C	C3		
020D	46		INC SI
020E	EB		JMP 205
020F	F5		
0210	88	DONE:	MOV [400],BL
0211	1E		
0212	00		
0213	04		
0214	CD		INT 20H
0215	20		

Program 6-2 is the DEBUG version. Run it with the following data:

```
[0420] = 07
[0421] = 23
[0422] = 31
[0423] = 20
[0424] = 16
[0425] = 38
[0426] = 00
```

RESULT:

```
[0400] = C9
```

What happens if you set [0423] = 00? What are the pros and cons of using a terminator, as compared with counting the elements? Which approach gives faster programs? Which makes data entry simpler?

Problem 6-10

If zero is an acceptable data value, you must use a different terminator. Revise Program 6-2 to use FF as a terminator. Run the revision with the original sample data except set [0426] = FF. Which terminator should you use if the data values are the numbers of characters received from a 30-cps (characters per second) teletypewriter in 1 s?

Like the countdown sequence, the sequence

```
MOV   AL,[SI]
INC   SI
```

has a single-instruction equivalent called LODSB. In fact, LODSB may either increment or decrement SI, depending on whether the D (direction) flag is 0 (increment) or 1 (decrement). The next program (Program 6-3 is the DEBUG version) is a revision of Program 6-2 to use LODSB. Note that we clear the D flag initially with CLD (the alternative is to set it with STD). In the flag register display, D appears as DN (1) or UP (0).

We call instructions like LODSB *string instructions*, as they handle arrays or strings of data elements. We call the automatic increase or decrease of SI an *autoincrement* or *autodecrement*, respectively. Note that LODSB always uses AL and SI.


```
        MOV     SI,420H     ;POINT TO START OF ARRAY
        SUB     BL,BL       ;SUM = ZERO INITIALLY
        CLD                 ;SELECT AUTOINCREMENTING
ADDELM: LODSB               ;GET NEXT ELEMENT
        TEST    AL,AL       ;IS IT TERMINATOR (ZERO)?
        JZ      DONE        ;YES, DONE
        ADD     BL,AL       ;NO, ADD ELEMENT TO SUM
        JMP     ADDELM
DONE:   MOV     [400H],BL   ;SAVE SUM
        INT     20H
```

PROGRAM 6-3

Memory Address (Hex)	Memory Contents (Hex)		Instruction (DEBUG Form)	
0200	*BE*		*MOV*	*SI,420*
0201	*20*			
0202	*04*			
0203	*28*		*SUB*	*BL,BL*
0204	*DB*			
0205	*FC*		*CLD*	
0206	*AC*	*ADDELM:*	*LODSB*	
0207	*84*		*TEST*	*AL,AL*
0208	*C0*			
0209	*74*		*JZ*	*20F*
020A	*05*			
020B	*00*		*ADD*	*BL,AL*
020C	*C3*			
020D	*EB*		*JMP*	*206*
020E	*F7*			
020F	*88*	*DONE:*	*MOV*	*[400],BL*
0210	*1E*			
0211	*00*			
0212	*04*			
0213	*CD*		*INT*	*20*
0214	*20*			

Limit Checking

We often want a computer to check whether data is valid, that is, whether it is within certain limits, below a threshold, or has an allowed value. Determining whether data is within limits is called *limit checking*. The key instruction here is

a comparison that subtracts a data value from a limit (or vice versa). If the operands are treated as unsigned, the Carry flag indicates which is larger. CMP op1,op2 sets the Carry as follows:

Carry = 0 if op1 $\geq$ op2
Carry = 1 if op1 $<$ op2

Carry is thus a *borrow*; that is, it is set if the difference is negative.

We can follow CMP with any of the following jump instructions:

JA (or JNBE): jump if above (Carry = 0 and Zero = 0).
JAE (JNB or JNC): jump if above or equal (Carry = 0).
JBE (or JNA): jump if below or equal (Carry = 1 or Zero = 1).
JB (JC or JNAE): jump if below (Carry = 1).

In general, after CMP op1,op2, the condition in the jump refers to how op1 compares to op2 (for example, is op1 "above" op2 ?). The alternative mnemonics simply make the sequence easier to understand.

The following program sums six elements but ignores ones that are 80 hex or above (that is, have a most significant bit of 1).

```
        MOV   CX,6        ;COUNT = NUMBER OF ELEMENTS
        MOV   SI,420H     ;POINT TO START OF ARRAY
        SUB   BL,BL       ;CLEAR SUM INITIALLY
        CLD               ;SELECT AUTOINCREMENTING
ADDELM: LODSB             ;COMPARE ELEMENT TO
                          ;  THRESHOLD
        CMP   AL,80H
        JAE   COUNT       ;BRANCH IF ABOVE OR AT
                          ;  THRESHOLD
        ADD   BL,AL       ;ELSE ADD ELEMENT TO SUM
COUNT:  LOOP  ADDELM      ;COUNT ELEMENTS
        MOV   [400H],AL   ;SAVE SUM
        INT   20H
```

Remember that LODSB increments SI automatically and LOOP decrements CX automatically.

Program 6-4 is the DEBUG version; run it with the following data:

[0420] = 07
[0421] = 20
[0422] = F1
[0423] = 3C
[0424] = 80
[0425] = 73

RESULT:

[0400] = D6

To make the threshold itself valid (that is, ignore elements that are above 80 hex, instead of 80 hex or above), simply replace JAE COUNT with JA COUNT.

Problem 6-11

Make Program 6-4 ignore elements that are 80 hex or above or 20 hex or below.

SAMPLE PROBLEM:

[0420] = 07
[0421] = 20
[0422] = F1
[0423] = 3C
[0424] = 80
[0425] = 73

RESULT:

[0400] = AF

PROGRAM 6-4

Memory Address (Hex)	Memory Contents (Hex)	Instruction (DEBUG Form)
0200	*B9*	*MOV CX,6*
0201	*06*	
0202	*00*	
0203	*BE*	*MOV SI,420*
0204	*20*	
0205	*04*	

PROGRAM 6-4 (continued)

Memory Address (Hex)	Memory Contents (Hex)		Instruction (DEBUG Form)	
0206	28		SUB	BL,BL
0207	DB			
0208	FC		CLD	
0209	AC	ADDELM:	LODSB	
020A	3C		CMP	AL,80
020B	80			
020C	73		JAE	210
020D	02			
020E	00		ADD	BL,AL
020F	C3			
0210	E2	COUNT:	LOOP	209
0211	F7			
0212	88		MOV	[400],BL
0213	1E			
0214	00			
0215	04			
0216	CD		INT	20
0217	20			

Limit checking is often used to reject measurements that are far removed from the majority (and therefore suspect). Many data analysis routines discard the highest and lowest values before computing statistics to eliminate readings caused by noise, equipment malfunction, or human error. Limit checking also guards against obviously incorrect entries, such as a time of 8:80 instead of the intended 8:00 or an automobile speed of 1000 km/hr instead of the intended 100 km/hr.

Displaying a Message

The next program (see Figure 6-4 for a flowchart and Program 6-5 for a DEBUG version) prints a message on the video display. It takes the message from locations starting at 420. Note that the delay loop uses register AX here.

```
        MOV     AH,0FH      ;CLEAR THE DISPLAY
        INT     10H
        SUB     AH,AH
        INT     10H
        MOV     DH,12       ;ROW 12
        MOV     DL,30       ;COLUMN 30
        SUB     BH,BH
        MOV     AH,2        ;MOVE CURSOR
        INT     10H
        CLD                 ;SELECT AUTOINCREMENTING
        MOV     SI,420H     ;POINT TO START OF MESSAGE
        MOV     CX,13       ;GET MESSAGE LENGTH
WRCHAR: LODSB               ;GET NEXT CHARACTER
        MOV     AH,0E       ;WRITE CHARACTER AND
        INT     10H         ;  ADVANCE CURSOR
        MOV     AX,0        ;WAIT A WHILE
DLY:    DEC     AX
        JNZ     DLY
        LOOP    WRCHAR      ;COUNT CHARACTERS
        INT     20H
```

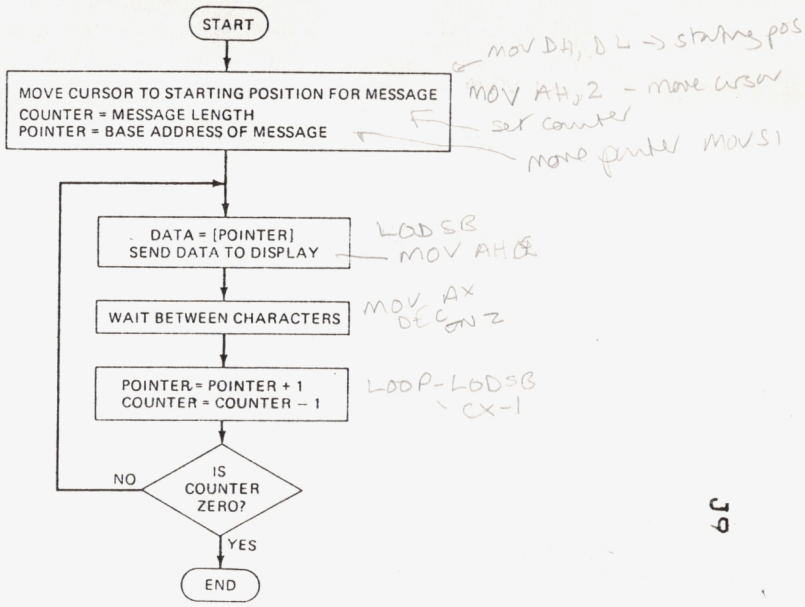


Figure 6-4
Flowchart for displaying a message.

PROGRAM 6-5

Memory Address (Hex)	Memory Contents (Hex)		Instruction (DEBUG Form)	
0200	B4		MOV	AH,F
0201	0F			
0202	CD		INT	10
0203	10			
0204	28		SUB	AH,AH
0205	E4			
0206	CD		INT	10
0207	10			
0208	B6		MOV	DH,C
0209	0C			
020A	B2		MOV	DL,1E
020B	1E			
020C	28		SUB	BH,BH
020D	FF			
020E	B4		MOV	AH,2
020F	02			
0210	CD		INT	10
0211	10			
0212	FC		CLD	
0213	BE		MOV	SI,420
0214	20			
0215	04			
0216	B9		MOV	CX,D
0217	0D			
0218	00			
0219	AC	WRCHAR:	LODSB	
021A	B4		MOV	AH,E
021B	0E			
021C	CD		INT	10
021D	10			
021E	B8		MOV	AX,0
021F	00			
0220	00			
0221	48	DLY:	DEC	AX
0222	75		JNZ	221
0223	FD			
0224	E2		LOOP	219
0225	F3			
0226	CD		INT	20
0227	20			

Run Program 6-5 with the following data:

```
[0420] = 41 A
[0421] = 42 B
[0422] = 41 A
[0423] = 4E N
[0424] = 44 D
[0425] = 4F O
[0426] = 4E N
[0427] = 20 _
[0428] = 53 S
[0429] = 48 H
[042A] = 49 I
[042B] = 50 P
[042C] = 21 !
```

You can use the E command to enter ASCII characters into memory. Just put quotation marks around them. For example, to put THE COMPUTER IS DEAD! in locations 420 through 434, type

E 420 "THE COMPUTER IS DEAD!"

You can also use the F (fill) command with the L option to fill memory with specific values or characters. For example, the following command clears locations 420 through 42F:

F 420 L 10 0

The order is: starting address, L, length in bytes, and fill value. The following command puts ASCII spaces in locations 420 through 44F:

F 420 L 30 " "

You can replace " " with 20 (an ASCII space).

Problem 6-12

Reduce the more significant byte of the delay count (location 220) according to the following sequence: 80, 40, 20, 10, 8, 4, 2, 1. Explain what happens.

Problem 6-13

Write a program that creates a "newspanel" or "Times Square" display in which the message appears to move from right to left. Your program should:

1. Start by clearing the entire display.
2. After a delay, put the first character of the message in the rightmost position.
3. Continue this process until the message has moved all the way across the display. Then start over.

Your data should consist of a set of blanks, the message, and another set of blanks. Both sets of blanks should be the same length as the message. Save an overall counter in register BP and a starting pointer in DI. The counter's initial value should be twice the message length plus 1 to allow for the leading blanks. The pointer should always contain the base address of the display data for the current iteration. Try the following message (13 characters long):

```
[0420]-[042C]= "  " (blanks)
[042D]-[0439]= "ABANDON SHIP!"
[043A]-[0446]= "  " (blanks)
```

One problem here is the static caused by the moving cursor. To get rid of it, blank the cursor by putting the following sequence ahead of your program:

```
MOV   AH,3              ;GET CURSOR CHARACTERISTICS
INT   10H
OR    CH,00100000B      ;BLANK CURSOR
MOV   AH,1
INT   10H
```

Then put the following sequence at the end to restore the cursor:

```
MOV   AH,3              ;GET CURSOR CHARACTERISTICS
INT   10H
AND   CH,11011111B      ;UNBLANK CURSOR
MOV   AH,1
INT   10H
```

If you forget to restore the cursor or accidentally return to DEBUG or DOS without it, enter and run the unblanking sequence. While DEBUG is tolerable without a cursor, your word processor will become very difficult to use (guess how I discovered this!).

Occasionally, for reasons that are beyond my comprehension, the cursor may end up as a solid block. If this happens, run the following program to get the underscore:

```
MOV   AH,3              ;GET CURSOR CHARACTERISTICS
INT   10H
AND   CH,00000111H      ;MAKE CURSOR ONE COLUMN HIGH
MOV   CH,CL
DEC   CH
MOV   AH,1
INT   10H
INT   3
```

As Table A2-4 shows, the IBM PC's video display can form many symbols besides standard typewriter characters. Among them are card suit symbols, graphics characters (parts of lines, circles, and solid figures), modified letters used in foreign languages such as French and German, mathematical symbols, and Greek letters. For example, you can use the following command to enter a cheerful message for Program 6-5:

```
E 420 1 "HAVE A NICE DAY!" 1
```

The message is 20 characters (14 hex) long, including the spaces. What happens if you replace the 1's with 2's?

Problem 6-14

Use Program 6-5 and the graphics characters DC through DF to draw a thick-sided box near the center of the screen. Use the line feed (0A), backspace (08), and space (20) characters to move the cursor.

Key Point Summary

1. Arrays are sets of elements with similar meanings or purposes. Each element is characterized by its number or index, and the entire array is characterized by its base address. Thus, to reach a particular element of an array, you must know the base address and the index.

2. The keys to processing arrays are:
   - A pointer that holds the address of the element being processed.
   - A flexible addressing mode that allows a single set of instructions to handle any element.
   - A counter or terminator that can be used to determine the length of the array.
3. To process arrays with the 8088, you can use an index register or base register to hold the pointer, register indirect addressing to reach the data in memory, and a data register to hold the counter or terminator.
4. The 8088 has special instructions for loop control and array or string operations. These instructions automatically perform overhead functions such as decrementing counters and incrementing or decrementing pointers. They apply only to specific registers.
5. Comparison instructions can determine whether an element is within limits. If the operands are unsigned, the Carry flag indicates which is larger.
6. Loops inside loops (that is, nested loops) and variable pointers and counters can handle multidimensional arrays and provide greater flexibility.

LABORATORY 9

Arithmetic

Purpose To learn how to do arithmetic with the 8088 microprocessor.

What You Should Learn

1. The standard BCD representation.
2. How to add and subtract decimal numbers.
3. How to add and subtract 16-bit numbers.
4. How to add and subtract multiple-precision numbers.
5. How to do arithmetic with lookup tables.

Reference Materials

Andrews, M. "Mathematical Microprocessor Software: A Square Root Comparison," *IEEE Micro*, May 1982, pp. 63–79.

Eccles, W. J. *Microprocessor Systems: A 16-Bit Approach*. Reading, MA: Addison-Wesley, 1985, pp. 92–95, 297–307, 324–326.

Gorsline, G. W. *16-Bit Modern Microcomputers*. Englewood Cliffs, NJ: Prentice-Hall, 1985, pp. 113–124 (arithmetic instructions), 124–125 (status flag instructions).

Hwang, K. *Computer Arithmetic*. New York: John Wiley, 1979.

Kulisch, U. W., and W. L. Mirankar. *Computer Arithmetic in Theory and Practice*. Orlando, FL: Academic Press, 1980.

Liu, Y. C., and G. A. Gibson, *Microcomputer Systems: The 8086/8088 Family* (2nd ed.). Englewood Cliffs, NJ: Prentice-Hall, 1986, pp. 63–78 (arithmetic instructions).

"Binary Floating-Point Arithmetic, Standard for (IEEE Std 754-1985)." IEEE, Piscataway, NJ, 1985.

Terms

Auxiliary carry: *see* **Half carry.**

BCD (binary-coded-decimal): a representation of decimal numbers in which each digit is coded separately in binary.

Carry: a bit that is 1 if an addition overflows into the succeeding digit position.

Half carry (or **auxiliary carry**)**:** the carry from the less significant 4 bits in an 8-bit operation.

Interpolation: estimating a function at points between those at which its values are known.

Linearization: approximating a function by a straight line between two points at which its values are known.

Pseudo-operation: an assembly language operation code that directs the assembler to do something but does not generate a machine language instruction. Also called a *directive* or *pseudo-instruction*.

8088 Instructions

ADC: add with carry. ADC op1,op2 produces the result op1 = op1 + op2 + Carry.

CLC: clear Carry flag (make it 0).

DAA: decimal adjust for addition; convert a binary sum in register AL to a BCD sum. DAA works only after an ADC AL or ADD AL instruction.

DAS: decimal adjust for subtraction; convert a binary difference in register AL to a BCD difference. DAS works only after an SBB AL or SUB AL instruction.

SBB: subtract with borrow. SBB op1,op2 produces the result op1 = op1 − op2 − Carry.

STC: set Carry flag to 1.

SUB: subtract. SUB op1,op2 produces the result op1 = op1 − op2.

8088 Assembler Pseudo-Operations

DB: define byte-length data; place 8-bit data items (separated by commas) in the next available memory locations. DB loads memory with fixed data (such as tables, messages, and numerical constants) needed for program execution.

DW: define word-length data; place 16-bit data items (separated by commas) in the next available memory locations. DW loads memory with 16-bit fixed data or addresses stored in the usual 8088 format (that is, less significant byte first).

ORG: set origin; place the object code generated from the subsequent statements in memory addresses, starting with the one specified. ORG lets the programmer determine where programs and data are placed in memory.

Applications of Arithmetic

Data processing almost always involves arithmetic. Typical operations are averaging, scaling, linearizing inputs, computing numerical integrals and derivatives, determining frequency responses, doing statistical analysis, and preparing plots. Simple applications require only binary or decimal addition and subtraction. Decimal arithmetic is necessary in calculators, business equipment, terminals, instruments, appliances, and games.

This laboratory starts with 8-bit binary arithmetic programs from Laboratory 6. It then covers decimal arithmetic, multibyte arithmetic, and the use of lookup tables.

8-Bit Binary Sum

The following program adds 8-bit unsigned binary numbers from 340 and 341 and puts the sum in 400.

```
MOV   AL,[340H]     ;GET FIRST NUMBER
ADD   AL,[341H]     ;ADD SECOND NUMBER
MOV   [400H],AL     ;SAVE SUM
INT   3             ;EXIT AND DISPLAY REGISTERS
```

Program 9-1 is the DEBUG version; we have inserted a NOP to allow for BCD operands later.

PROGRAM 9-1

Memory Address (Hex)	Memory Contents (Hex)	Instruction (DEBUG Form)
0200	*A0*	*MOV AL,[340]*
0201	*40*	
0202	*03*	
0203	*02*	*ADD AL,[341]*
0204	*06*	

PROGRAM 9-1 (continued)

Memory Address (Hex)	Memory Contents (Hex)	Instruction (DEBUG Form)
0205	*41*	
0206	*03*	
0207	*90*	*NOP*
0208	*A2*	*MOV [400],AL*
0209	*00*	
020A	*04*	
020B	*CC*	*INT 3*

Run Program 9-1 for the following cases:

1. [0340] = 32
   [0341] = 25
   Result: [0400] = 57
2. [0340] = 38
   [0341] = 25
   Result: [0400] = 5D

You can use the H (hexarithmetic) command to check the sums.

Problem 9-1

Make Program 9-1 subtract instead of add. Try the following cases:

a. [0340] = 32
   [0341] = 25
   Result: [0400] = 0D

b. [0340] = 32
   [0341] = 58
   Result: [0400] = DA

Problem 9-2

Make Program 9-1 save the carry in 401. Try the following cases:

a. [0340] = 38
   [0341] = 25
   Result: [0400] = 5D
   [0401] = 00

b. [0340] = 98
[0341] = 89
Result: [0400] = 21
[0401] = 01

(*Hint:* Use ADC AH,0 to add the Carry to AH.)

Binary-Coded-Decimal (BCD) Representation

A BCD code is the simplest way to represent decimal numbers in a computer, because it does not require multiplications or divisions by 10. In the standard BCD code (Table 9-1), 0 through 9 are the same as in binary. However, numbers above 9 are different (see Table 9-2 for examples). Note the following:

1. Each decimal digit is encoded separately in BCD. This is not true in binary, since 10 is not an integral power of 2. When computers take over the world, they will force people to have either 8 or 16 fingers.
2. The BCD representation takes more memory than the binary representation. For example, 8 bits can represent a binary number as large as 255 but only 99 in BCD. The number 999 requires three BCD digits (12 bits) but only 10 bits in binary (because $2^{10} = 1024$).
3. Some binary numbers are invalid in BCD. In standard BCD, no digit can exceed 9.

Table 9-1
Standard BCD Representation

Decimal Digit	*BCD Representation*
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Table 9-2
Standard BCD Representations of Some Decimal Numbers

Decimal Number	*BCD Representation*	*Binary Representation*
10	0001 0000	00001010
11	0001 0001	00001011
12	0001 0010	00001100
13	0001 0011	00001101
16	0001 0110	00010000
25	0010 0101	00011001
50	0101 0000	00110010
66	0110 0110	01000010
83	1000 0011	01010011

BCD numbers are difficult to process in binary arithmetic units. The reason is that BCD 10 (00010000) is not one larger than BCD 9 (00001001); it is, in fact, seven larger. (Try subtracting!) Thus, to get a BCD sum from a binary adder, you must add an extra 6 whenever the sum of two digits exceeds 9. Subtraction works similarly: You must subtract an extra 6 whenever the difference between two digits is negative.

EXAMPLE 1:

```
  33 (BCD) = 00110011
+ 25 (BCD) = 00100101
             01011000 = 58 (BCD)
```

There is no problem here, as neither digit exceeds 9.

EXAMPLE 2:

```
  38 (BCD) = 00111000
+ 25 (BCD) = 00100101
             01011101 = 5D
```

Here we need an extra 6 in the lower digit position, as 8 + 5 produces a decimal carry.

```
  5D
+ 06
  63
```

EXAMPLE 3:

```
   98 (BCD) = 10011000
+  25 (BCD) = 00100101
              10111101 = BD
```

Here we need an extra 6 in both digit positions.

```
   BD
+  66
  123
```

Obviously, a program that decides when to add or subtract 6 is not easy to write, as it must check each digit. Because decimal arithmetic is essential to many common applications, most processors have special instructions for it. The 8088 has DAA (decimal adjust for addition), and DAS (decimal adjust for subtraction). They convert a binary sum or difference in AL (the result of an ADC AL, ADD AL, SBB AL, or SUB AL instruction) to a decimal sum or difference; that is, DAA adds 6 and DAS subtracts 6 wherever necessary.

8-Bit Decimal Sum

The following program adds BCD numbers from 340 and 341 and saves the sum in 400.

```
MOV   AL,[340H]    ;GET FIRST NUMBER
ADD   AL,[341H]    ;ADD SECOND NUMBER
DAA
MOV   [400H],AL    ;SAVE SUM
INT   3
```

The only change from Program 9-1 is DAA instead of NOP.

Run the BCD version of Program 9-1 with the following data:

1. [0340] = 32
   [0341] = 25
   Result: [0400] = 57
2. [0340] = 38
   [0341] = 25
   Result: [0400] = 63

The second sum differs from the binary result.

Problem 9-3

In Program 9-1, what do register AL, the Auxiliary Carry flag, and the Carry flag contain after ADD AL,[341H] for the following examples:

a. [0340] = 38
   [0341] = 25
b. [0340] = 98
   [0341] = 25
c. [0340] = 98
   [0341] = 89
d. [0340] = 90
   [0341] = 91

To display the registers and flags, put a breakpoint in location 207. Note (see Table 2-3) that DEBUG displays the Auxiliary Carry as AC (1) or NA (0). Why is the Auxiliary Carry necessary? (*Hint:* Examine the results of examples *c* and *d* above.)

Problem 9-4

Write a decimal subtraction program and try it on the examples in Problem 9-3. What is Carry at the end of each example? What does Carry mean at the end?

Problem 9-5

Extend the decimal addition program to show the sum near the center of the screen.

Decimal Summation

The following revision of Program 6-1 adds an array of unsigned 8-bit binary numbers, starting at 340; it puts the sum in 400. The length of the array is in 401. Program 9-2 is the DEBUG version.

```
        MOV   CL,[401H]    ;COUNT = ARRAY LENGTH
        SUB   CH,CH        ;EXTEND COUNT TO 16 BITS
        MOV   SI,340H      ;POINT TO BASE OF ARRAY
        SUB   AL,AL        ;SUM = ZERO INITIALLY
ADDELM: ADD   AL,[SI]      ;ADD ELEMENT TO SUM
        NOP
```

```
        INC     SI              ;CONTINUE THROUGH ALL
                                ;  ELEMENTS
        LOOP    ADDELM
        MOV     [400H],AL       ;SAVE SUM
        INT     20H
```

PROGRAM 9-2

Memory Address (Hex)	Memory Contents (Hex)		Instruction (DEBUG Form)	
0200	*8A*		*MOV*	*CL,[401]*
0201	*0E*			
0202	*01*			
0203	*04*			
0204	*28*		*SUB*	*CH,CH*
0205	*ED*			
0206	*BE*		*MOV*	*SI,340*
0207	*40*			
0208	*03*			
0209	*28*		*SUB*	*AL,AL*
020A	*C0*			
020B	*02*	*ADDELM:*	*ADD*	*AL,[SI]*
020C	*04*			
020D	*90*		*NOP*	
020E	*46*		*INC*	*SI*
020F	*E2*		*LOOP*	*20B*
0210	*FA*			
0211	*A2*		*MOV*	*[400],AL*
0212	*00*			
0213	*04*			
0214	*CD*		*INT*	*20*
0215	*20*			

We have again inserted a NOP to allow a BCD version. Run Program 9-2 with the following data:

```
[0401] = 03     (number of elements)
[0340] = 35     (elements)
[0341] = 47
[0342] = 28
```

RESULT:

```
[0400] = A4
```


Change the program to make the summation decimal (BCD) rather than binary. Run the revision with the same data. The answer should be

```
[0400] = 10
```

16-Bit Arithmetic

We can easily extend Program 9-2 to handle 16-bit numbers. The only changes are the use of AX instead of AL and two increments of SI instead of one. Of course, the sum now occupies two bytes of memory. The revised program is as follows (Program 9-3 is the DEBUG version):

```
        MOV     CL,[402H]       ;COUNT = ARRAY LENGTH
        SUB     CH,CH           ;EXTEND COUNT TO 16 BITS
        MOV     SI,340H         ;POINT TO BASE OF ARRAY
        SUB     AX,AX           ;SUM = ZERO INITIALLY
ADDELM: ADD     AX,[SI]         ;ADD ELEMENT TO SUM
        INC     SI
        INC     SI
        LOOP    ADDELM
        MOV     [400H],AX       ;SAVE SUM
        INT     20H
```

Run Program 9-3 with the following data:

```
[0402] = 02     (number of 16-bit elements)
[0340] = 3E     (LSBs of first element)
[0341] = 47     (MSBs of first element)
[0342] = F5     (LSBs of second element)
[0343] = 2A     (MSBs of second element)
```

RESULT:

```
[0400] = 33     (LSBs of sum)
[0401] = 72     (MSBs of sum)
```

that is,

```
   473E
 + 2AF5
 ------
   7233
```

PROGRAM 9-3

Memory Address (Hex)	Memory Contents (Hex)		Instruction (DEBUG Form)	
0200	*8A*		*MOV*	*CL,[402]*
0201	*0E*			
0202	*02*			
0203	*04*			
0204	*28*		*SUB*	*CH,CH*
0205	*ED*			
0206	*BE*		*MOV*	*SI,340*
0207	*40*			
0208	*03*			
0209	*29*		*SUB*	*AX,AX*
020A	*C0*			
020B	*03*	*ADDELM:*	*ADD*	*AX,[SI]*
020C	*04*			
020D	*46*		*INC*	*SI*
020E	*46*		*INC*	*SI*
020F	*E2*		*LOOP*	*20B*
0210	*FA*			
0211	*A3*		*MOV*	*[400],AX*
0212	*00*			
0213	*04*			
0214	*CD*		*INT*	*20*
0215	*20*			

A decimal version is complicated by the fact that DAA and DAS work only on bytes in AL. The program thus must handle the data a byte at a time. However, it must also deal with carries between bytes. Here ADC (add with Carry) is useful, as its result is

Operand 1 = Operand 1 + Operand 2 + Carry

The way to do 16-bit decimal addition is, therefore,

1. Add the less significant bytes using ADD and DAA.
2. Add the more significant bytes using ADC and DAA.

The following program adds an array of 16-bit decimal numbers, starting at 340; it puts the sum in 400 and 401. Each number occupies two memory locations, with the less significant byte first. The length of the array (how many 16-bit numbers there are) is in 402.


```
        MOV     CL,[402H]   ;COUNT = ARRAY LENGTH
        SUB     CH,CH       ;EXTEND COUNT TO 16 BITS
        MOV     SI,340H     ;POINT TO BASE OF ARRAY
        SUB     BX,BX       ;SUM = ZERO INITIALLY
        CLD                 ;SELECT AUTOINCREMENTING
ADDELM: LODSW               ;GET NEXT 16-BIT ELEMENT
        ADD     AL,BL       ;ADD LESS SIGNIFICANT BYTES
        DAA                 ;MAKE SUM DECIMAL
        MOV     BL,AL
        MOV     AL,AH       ;ADD MORE SIGNIFICANT BYTES
        ADC     AL,BH
        DAA                 ;MAKE SUM DECIMAL
        MOV     BH,AL
        LOOP    ADDELM
        MOV     [400H],BX   ;SAVE SUM
        INT     20H
```

Program 9-4 is the DEBUG version. Run it with the following data:

```
[0402] = 02   (number of four-digit elements)

[0340] = 36   (LSDs of first element)
[0341] = 21   (MSDs of first element)
[0342] = 97   (LSDs of second element)
[0343] = 18   (MSDs of second element)
```

RESULT:

```
[0400] = 33   (LSDs of sum)
[0401] = 40   (MSDs of sum)
```

that is,

```
  2136
+ 1897
------
  4033
```

PROGRAM 9-4

Memory Address (Hex)	Memory Contents (Hex)		Instruction (DEBUG Form)	
0200	*8A*		*MOV*	*CL,[402]*
0201	*0E*			
0202	*02*			
0203	*04*			
0204	*28*		*SUB*	*CH,CH*
0205	*ED*			
0206	*BE*		*MOV*	*SI,340*
0207	*40*			
0208	*03*			
0209	*29*		*SUB*	*BX,BX*
020A	*DB*			
020B	*FC*		*CLD*	
020C	*AD*	*ADDELM:*	*LODSW*	
020D	*00*		*ADD*	*AL,BL*
020E	*D8*			
020F	*27*		*DAA*	
0210	*88*		*MOV*	*BL,AL*
0211	*C3*			
0212	*88*		*MOV*	*AL,AH*
0213	*E0*			
0214	*10*		*ADC*	*AL,BH*
0215	*F8*			
0216	*27*		*DAA*	
0217	*88*		*MOV*	*BH,AL*
0218	*C7*			
0219	*E2*		*LOOP*	*20C*
021A	*F1*			
021B	*89*		*MOV*	*[400],BX*
021C	*1E*			
021D	*00*			
021E	*04*			
021F	*CD*		*INT*	*20*
0220	*20*			

Problem 9-6

Extend Program 9-4 to place the carries in 402. Use 403 for the number of elements. Try the following example:

[0403] = 02 (number of four-digit elements)

[0340] = 36 (LSDs of first element)
[0341] = 21 (MSDs of first element)
[0342] = 97 (LSDs of second element)
[0343] = 98 (MSDs of second element)

RESULT:

[0400] = 33 (LSDs of sum)
[0401] = 20 (middle digits of sum)
[0402] = 01 (MSDs of sum = carries)

that is,

```
  2136
+ 9897
 12033
```

Be sure to keep the carries as a decimal number. That is, do not simply use INC to add a Carry to the most significant digits.

Multiple-Precision Arithmetic

We can extend Programs 9-1 through 9-4 to handle numbers of any length. The procedure (see Figure 9-1) is as follows:

1. Initialization.

COUNT = LENGTH OF NUMBERS (IN BYTES)
POINTER 1 = BASE ADDRESS OF NUMBER 1
POINTER 2 = BASE ADDRESS OF NUMBER 2
CARRY = 0, as there is never a Carry into the least significant bytes.

2. Add bytes.

[POINTER 1] = [POINTER 1] + [POINTER 2] + CARRY

This step generates a new CARRY.

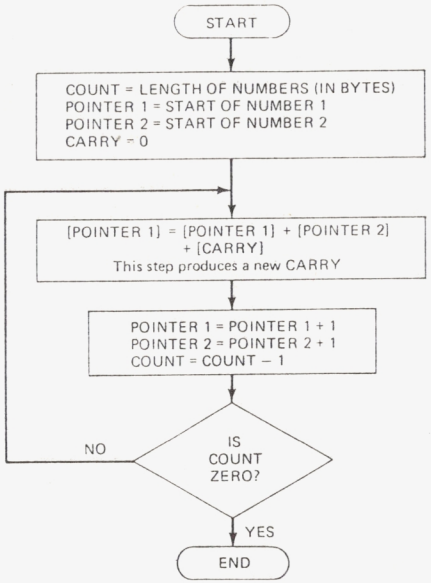

START
COUNT = LENGTH OF NUMBERS (IN BYTES)
POINTER 1 = START OF NUMBER 1
POINTER 2 = START OF NUMBER 2
CARRY = 0
[POINTER 1] = [POINTER 1] + [POINTER 2] + [CARRY]
This step produces a new CARRY
POINTER 1 = POINTER 1 + 1
POINTER 2 = POINTER 2 + 1
COUNT = COUNT − 1
IS COUNT ZERO?
NO
YES
END

Figure 9-1
Flowchart for multiple-precision addition.

3. Update counter and pointers.

POINTER 1 = POINTER 1 + 1
POINTER 2 = POINTER 2 + 1
COUNT = COUNT − 1

If COUNT is not 0, return to Step 2.

The numbers are in the usual 8088 format with their least significant bytes at the lowest addresses.

If the length of the numbers is in 400, the numbers start in 340 and 360, and the sum replaces the number starting in 340, the program is

```
        MOV     CL,[400H]       ;GET LENGTH OF NUMBERS IN
                                ;  BYTES
        SUB     CH,CH           ;EXTEND LENGTH TO 16 BITS
        MOV     DI,340H         ;POINT TO BASE OF FIRST
                                ;  NUMBER
        MOV     SI,360H         ;POINT TO BASE OF SECOND
                                ;  NUMBER
        CLD                     ;SELECT AUTOINCREMENTING
        CLC                     ;CLEAR CARRY INITIALLY
ADBYTE: LODSB                   ;GET BYTE OF SECOND
                                ;  NUMBER
        ADC     AL,[DI]         ;ADD BYTE OF FIRST NUMBER
        STOSB                   ;STORE SUM AS FIRST NUMBER
        LOOP    ADBYTE
        INT     20H
```


Program 9-5 is the DEBUG version. A key factor is that LODSB, STOSB, and LOOP do not affect Carry, so its value is available for the next ADC. INC and DEC do not affect the Carry either. Thus they can also be used in loops that depend on Carry to save a bit for the next iteration.

PROGRAM 9-5

Memory Address (Hex)	Memory Contents (Hex)		Instruction (DEBUG Form)	
0200	*8A*		*MOV*	*CL,[400]*
0201	*0E*			
0202	*00*			
0203	*04*			
0204	*28*		*SUB*	*CH,CH*
0205	*ED*			
0206	*BF*		*MOV*	*DI,340*
0207	*40*			
0208	*03*			
0209	*BE*		*MOV*	*SI,360*
020A	*60*			
020B	*03*			
020C	*FC*		*CLD*	
020D	*F8*		*CLC*	
020E	*AC*	*ADBYTE:*	*LODSB*	
020F	*12*		*ADC*	*AL,[DI]*
0210	*05*			
0211	*AA*		*STOSB*	
0212	*E2*		*LOOP*	*20E*
0213	*FA*			
0214	*CD*		*INT*	*20*
0215	*20*			

Try Program 9-5 on the following example:

```
[0400] = 06     (length of numbers in bytes)

[0340] = C7     (LSBs of first number)
[0341] = 59
[0342] = F0
[0343] = AB     (MSBs of first number)

[0360] = EA     (LSBs of second number)
[0361] = 93
[0362] = A1
[0363] = 28     (MSBs of second number)
```

RESULT:

```
[0340] = B1     (LSBs of sum)
[0341] = ED
[0342] = 91
[0343] = D4     (MSBs of sum)
```

that is,

```
   ABF059C7
+  28A193EA
   D491EDB1
```

Problem 9-7

Write a program that adds decimal numbers of arbitrary length. Assume the same conditions as in Program 9-5. Try the following sample case:

```
[0400] = 04     (length of numbers in bytes)

[0340] = 60     (LSDs of first number)
[0341] = 71
[0342] = 34
[0343] = 29     (MSDs of first number)

[0360] = 28     (LSDs of second number)
[0361] = 81
[0362] = 93
[0363] = 15     (MSDs of second number)
```


RESULT:

```
[0340] = 88     (LSDs of sum)
[0341] = 52
[0342] = 28
[0343] = 45     (MSDs of sum)
```

that is,

```
   29347160
+  15938128
   45285288
```

Problem 9-8

Write a program that subtracts decimal numbers of arbitrary length. Assume the same conditions as in Program 9-5. Subtract the number starting at 360 from the one starting at 340. Try the following example:

```
[0400] = 04     (length of numbers in bytes)

[0340] = 60     (LSDs of minuend)
[0341] = 71
[0342] = 34
[0343] = 29     (MSDs of minuend)

[0360] = 81     (LSDs of subtrahend)
[0361] = 28
[0362] = 60
[0363] = 19     (MSDs of subtrahend)
```

RESULT:

```
[0340] = 79     (LSDs of difference)
[0341] = 42
[0342] = 74
[0343] = 09     (MSDs of difference)
```

that is,

```
   29347160
-  19602881
   09744279
```

Arithmetic with Lookup Tables

One way to do arithmetic is by using lookup tables that contain all possible results. The program must then locate the one we want, just as in the code conversion of Program 5-5.

For example, suppose we form a table of the squares of the decimal digits. The following program uses it to square the digit in 400, placing the square in 401.

```
MOV   AL,[400H]      ;GET DATA
MOV   BX,380H        ;GET BASE ADDRESS OF TABLE
XLAT                 ;GET SQUARE FROM TABLE
MOV   [401H],AL      ;SAVE SQUARE
INT   20H

ORG   380H           ;SQUARES OF DECIMAL DIGITS
DB    0,1,4,9,16,25,36,49,64,81
```

ORG (Set Origin) is an assembler directive (called a *pseudo-operation*); it indicates where the object code generated from subsequent statements is to be placed in memory. The directive DB (Define Byte-Length Data) places a list of 8-bit data items in the next available locations.

Program 9-6 is the DEBUG version. Remember that the table entries are in hex (in case you are wondering why $4^2 = 10$!). You can use DB and DW (Define Word-Length Data) in DEBUG programs; both take commas between operands and ASCII characters inside quotation marks. However, because DEBUG does not allow ORG, you must exit from the current A command and start another at the new address to simulate its effect.

PROGRAM 9-6

Memory Address (Hex)	Memory Contents (Hex)	Instruction (DEBUG Form)
0200	*A0*	*MOV AL,[400]*
0201	*00*	
0202	*04*	
0203	*BB*	*MOV BX,380*
0204	*80*	
0205	*03*	
0206	*D7*	*XLAT*
0207	*A2*	*MOV [401],AL*
0208	*01*	

PROGRAM 9-6 (continued)

Memory Address (Hex)	Memory Contents (Hex)	Instruction (DEBUG Form)
0209	*04*	
020A	*CD*	*INT 20*
020B	*20*	
0380	*00*	*DB 0,*
0381	*01*	*1,*
0382	*04*	*4,*
0383	*09*	*9,*
0384	*10*	*10,*
0385	*19*	*19,*
0386	*24*	*24,*
0387	*31*	*31,*
0388	*40*	*40,*
0389	*51*	*51*

Run Program 9-6 for the following examples:

1. [0400] = 04
   Result: [0401] = 10 hex = 16 decimal
2. [0400] = 07
   Result: [0401] = 31 hex = 49 decimal

The table lookup saves time, even though the 8088 has a multiply (MUL) instruction. XLAT takes only 11 cycles, whereas MUL takes 70 to 77 cycles for an 8-bit operation.

Problem 9-9

Write a program that uses the table in Program 9-6 to add the squares of 400 and 401. Put the sum in 402.

EXAMPLE:

[0400] = 03
[0401] = 06

RESULT:

[0402] = 2D (hex), as 2D = 09 (3^2) + 24 (6^2). In decimal, this is 45 = 9 + 36.

Problem 9-10

Write a program that uses a table to cube a decimal digit. Allow two bytes for each entry, as some cubes exceed 256. Assume that the data is in 400 and put the result in 401 and 402 (MSBs in 402).

EXAMPLES:

1. [0400] = 03
   Result: [0401] = 1B
   [0402] = 00
2. [0400] = 07
   Result: [0401] = 57
   [0402] = 01

The results are hexadecimal numbers.

Problem 9-11

Write a program that takes the four-digit square root of a decimal digit. The digit is in 400 and the square root ends up in 401 and 402 (more significant digits in 402). Use the following table starting at 380; enter it with a DW pseudo-operation.

Value	*Square Root*
0	00.00
1	01.00
2	01.41
3	01.73
4	02.00
5	02.24
6	02.45
7	02.65
8	02.83
9	03.00

EXAMPLES:

1. [0400] = 03
   Result: [0401] = 73
   [0402] = 01
2. [0400] = 07
   Result: [0401] = 65
   [0402] = 02

Problem 9-12

Extend the answer to Problem 9-11 to produce a six-digit square root in 401, 402, and 403 (most significant digits in 403). Use the following table:

Value	*Square Root*
0	00.0000
1	01.0000
2	01.4142
3	01.7321
4	02.0000
5	02.2361
6	02.4495
7	02.6458
8	02.8284
9	03.0000

EXAMPLES:

1. [0400] = 02
   Result: [0401] = 42
   [0402] = 41
   [0403] = 01
2. [0400] = 06
   Result: [0401] = 95
   [0402] = 44
   [0403] = 02

If the table is long, one option is to keep only some entries in memory. You could, for example, keep every tenth entry and interpolate to get intermediate values.

Key Point Summary

1. BCD is a convenient way to represent decimal numbers, as each digit is coded separately. However, BCD requires more memory and processing instructions than the binary representation.
2. The 8088 processor has special decimal arithmetic instructions. DAA converts a byte-length binary sum to decimal, and DAS does the same to a byte-length binary difference. Both operate only on register AL; DAA works only after an ADC AL or ADD AL instruction, whereas DAS works only after an SBB AL or SUB AL instruction.
3. Multiple-precision arithmetic requires a series of 8-bit or 16-bit operations. The Carry flag transfers carries or borrows between them. Decimal operations must be done one byte at a time.
4. Lookup tables provide a simple way to do complex arithmetic. The lookup procedure depends only on the organization of the table and the length of the elements; it does not depend on the data values or the function involved.